

POLITECHNIKA ŁÓDZKA
Wydział Elektrotechniki, Elektroniki,
Informatyki i Automatyki

Praca dyplomowa magisterska

**Analiza problemu komiwojażera przy pomocy algorytmów
heurystycznych**

Traveling salesman problem analysis using heuristic algorithms

Łukasz Tracz

Nr albumu 234011

Opiekun pracy:

Dr inż. Paweł Marciniak

Opiekun dodatkowy:

Dr inż. Cezary Maj

Łódź, 2022

Streszczenie

Abstract

Spis treści

1. WSTĘP	5
1.1 GENEZA TSP	5
1.2 PROBLEMATYKA ZAGADNIENIA TSP	5
1.3 ZASTOSOWANIA TSP	8
1.3.1 Sekwencjonowanie DNA przez hybrydyzację	8
1.3.2 Optymalizacja ruchów robota do produkcji mikrochipów	9
1.4 TSP A METODY HEURYSTYCZNE	10
1.5 MOTYWACJA	12
2. CEL I ZAŁOŻENIA PRACY	13
3. OMÓWIENIE BADANYCH ALGORYTMÓW I BIBLIOTEK	15
3.1 BIBLIOTEKA „PYTHON-TSP”	15
3.1.1 Metoda siłowa	15
3.1.2 Algorytm Bellmana-Helda-Karpa	16
3.1.3 Algorytm symulowanego wyżarzania	17
3.1.3.1 Metalurgia, inspiracją dla naukowców	17
3.1.3.2 Ogólny zarys działania metaheurystyki	18
3.1.4 Algorytm przeszukiwania lokalnego	19
3.1.5 Cechy biblioteki python-tsp	20
3.2 ALGORYTM A*	21
3.2.1 Działanie algorytmu	21
3.2.2 Opis algorytmu A*	22
3.3 ALGORYTM NAJBLIŻSZEGO SĄSIADA (GREEDY SEARCH)	23
3.4 ALGORYTM GENETYCZNY	24
3.4.1 Terminologia	24
3.4.2 Schemat działania AG	25
3.4.3 Operatory genetyczne oraz kodowanie dla TSP	26
3.4.4 AG porównanie biblioteki mlrose i scikit-opt	29
3.5 ALGORYTM MRÓWKOWY	31
3.5.1 Opis algorytmu mrówkowego	32
3.5.2 ACO a problem TSP	33
3.6 ALGORYTM OPTYMALIZACYJNY Z WYKORZYSTANIEM ROJU	34
3.6.1 Schemat działania algorytmu PSO	34
3.6.2 Modyfikacja PSO dla problemu TSP	36

4. PLAN PRACY.....	38
4.1 PRZYGOTOWANIE DANYCH DO BADAŃ	39
4.2 METRYKI POMIAROWE	40
5. CELE BADAŃ	41
6. ARCHITEKTURA POMIAROWA	43
7. PRZEGLĄD WYNIKÓW BADAŃ	46
7.1 ANALIZA ALGORYTMU NAJBLIŻSZEGO SĄSIADA	46
7.2 ANALIZA ALGORYTMU A*	48
7.2.1 A* wyniki pomiarów zużycia CPU	49
7.2.2 A* analiza błędów uzyskanych rozwiązań	51
7.2.3 A* analiza zużycia pamięci RAM dla obu heurystyk	54
7.3 ANALIZA ALGORYTMU PRZESZUKIWAŃ LOKALNYCH.....	56
7.3.1 Analiza pomiarów efektywności i czasu wykonania	57
7.3.2 Analiza pomiarów zużycia CPU	59
7.4 ANALIZA WYNIKÓW ALGORYTMU MRÓWKOWEGO	60
7.4.1 ACO poszukiwanie optymalnej wartości RHO.....	61
7.4.2 ACO poszukiwanie wartości parametrów alfa i beta	63
7.5 ANALIZA WYNIKÓW ALGORYTMU ROJU CZĄSTEK (PSO).....	63
7.5.1 PSO poszukiwanie wartości parametrów alfa oraz beta	64
7.6 PSO ANALIZA CZASU WYKONANIA	68
8. KONKLUZJE	71
9. SPIS ILUSTRACJI	74
10. BIBLIOGRAFIA	76

1. Wstęp

Obecnie transport oraz logistyka to dwa pojęcia bez których świat, który znamy dzisiaj w ogóle nie mógłby powstać. Wszystkie państwa na świecie posiadają zawsze jakiś zasób w swoich granicach którego inny kraj w tej chwili wymaga. Taki stan rzeczy pozwolił na rozwój obecnie szeroko pojętego handlu, czyli realizacji wymiany dóbr na konkretną ilość danego środka płatniczego. Przedmioty transakcji, zazwyczaj wymagają dostarczenia ich do miejsca przeznaczenia w stosownym czasie i poniesienia jak najmniejszego kosztu związanego z ich transportem.

1.1 Geneza TSP

Problem komiwojażera (ang. Travelling Salesman Problem) to wyzwanie optymalizacji z ograniczeniami tzn. jak znaleźć najlepszy układ zbiorów zmiennych przy znanych regułach i parametrach punktacji. Pomimo że ten problem znany był matematykom od lat to zainteresowanie nim dopiero wzrosło od lat 30 XX wieku [1]. Jednym z pierwszych zainteresowanych tym problemem, który ponownie przedstawił to wyzwanie światu był Hassler Whitney z Harvardu, czego dokonał podczas swojego wykładu na Princeton University w 1934 r. Po jego wystąpieniu publiczność z USA zapamiętała wyzwanie jako „48 stanowy problem Hasslera Whitneya”, ponieważ jego wersja problemu zakładała znalezienie optymalnej trasy pozwalającej odwiedzić wszystkie stolice 48 stanów (Alaska i Hawaje wtedy jeszcze nie były stanami). Problem ten był jeszcze badany przez wielu zdolnych matematyków i doczekał się naprawę wiele publikacji naukowych.

1.2 Problematyka zagadnienia TSP

Cała trudność związana z problemem komiwojażera wiąże się z bardzo szybko rosnącą liczbą elementów znajdujących się w naszej przestrzeni rozwiązań wraz ze wzrostem ilości miast, które komiwojażer powinien odwiedzić. Przy założeniu, że rozważamy symetryczny problem komiwojażera (droga z punktu A do B jest taka sama jak z B do A) przy grafie posiadającym N wierzchołków, liczba wszystkich możliwych cykli Hamiltona wynosi $(N-1)! / 2$. Praktycznie sprawdzenie wszystkich tych kombinacji jest możliwe tylko wtedy, gdy graf składa się z niewielkiej liczby wierzchołków. Kolejnym ważnym aspektem jest fakt, że, dopóki cały zbiór wszystkich rozwiązań spełniających nasze założenia nie został przeszukany, nie jesteśmy w stanie jednoznacznie stwierdzić

czy optymalne rozwiązanie już zostało przez nas znalezione i powinniśmy przerwać poszukiwania. Najlepiej wzrost tej trudności opisuje poniższa tabela.

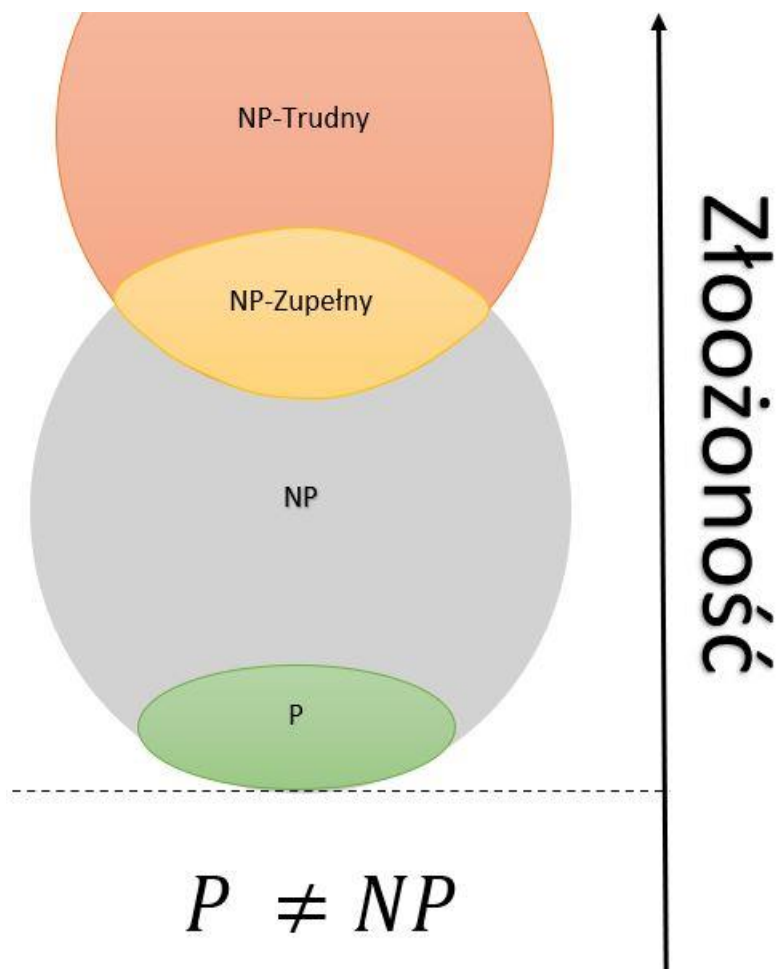
Tabela 1 Tabela przedstawiająca wzrost przestrzeni poszukiwań w zależności od ilości miast dostępnych do odwiedzenia dla komiwojażera

N Ilość miast do odwiedzenia	$\frac{(n-1)!}{2}$ Ilość wszystkich sekwencji spełniających założenia symetrycznego TSP
4	3
5	12
6	60
7	360
8	2 520
9	20 160
10	181 440
20	$\approx 6 * 10^{16}$

Z uwagi na opisane powyżej trudności, nie istnieje jeszcze ogólny i równocześnie efektywny algorytm zapewniający zawsze optymalne rozwiązanie dla TSP. Wówczas został on zaklasyfikowany do grupy problemów NP-trudnych. Z uwagi na to, że istnieje bardzo dużo problemów o złożoności czasowej większej niż wielomianowa, dla problemów NP. utworzono specjalne podzbiory, które umożliwiają dokładniejsze określenie złożoności problemu.

Poniższy rysunek ma za zadanie zwizualizować oraz wyjaśnić relacje łączące klasę problemu z jego złożonością obliczeniową.

- problem P – problem decyzyjny, dla którego w czasie wielomianowym jesteśmy w stanie znaleźć rozwiązanie
- problem NP – problem decyzyjny, gdzie znamy rozwiązanie problemu, jeżeli w czasie wielomianowym jesteśmy w stanie zweryfikować jego poprawność
- problem NP-zupełny – problem decyzyjny, dla którego niemożliwe jest znalezienie rozwiązania w czasie wielomianowym
- problem NP-trudny – problem obliczeniowy, dla którego nie jest możliwe znalezienie rozwiązania w czasie wielomianowym oraz weryfikacja jego poprawności jest równie trudna co każdego innego problemu z klasy NP.



Rysunek 1 Zależność problemów P, NP, NP-zupełnych i NP-trudnych [53]

1.3 Zastosowania TSP

Pomimo że omawiany problem łatwo odnieść do problemów logistycznych, to jego zastosowanie ma miejsce również w innych dziedzinach, takich jak biologia molekularna oraz produkcja mikrochipów.

1.3.1 Sekwencjonowanie DNA przez hybrydyzację

Przed przystąpieniem do omówienia zastosowania TSP w biologii molekularnej, należy wpierw wyjaśnić trzy terminy DNA oraz procesy sekwencjonowania i hybrydyzacji.

DNA jest to organiczny związek chemiczny, odkryty w 1962 r. przez Watsona i Crick, którzy wspólnie Wilkinsem, uzyskali Nagrodę Nobla za odkrycie molekularnej struktury kwasów nukleinowych i jej znaczenia dla przekazywania informacji w materii żywej [2]. Związek ten pełni rolę nośnika informacji genetycznej u organizmów żywych i wirusów. Jedną z najważniejszych cech DNA jest jego komplementarność tj. Adenina w DNA zawsze łączy się z Tyminom a Guanina z Cytosyną.

Sekwencjonowanie DNA jest to proces, mający na celu ustalenia kolejności nukleotydów / ciągów n nukleotydów wchodzących w skład poddanej sekwencjonowaniu nici DNA [3].

Hybrydyzacja jest to zjawisko spontanicznego łączenia się komplementarnych nici kwasów nukleinowych tj. (DNA z DNA, RNA z RNA lub DNA z RNA) [4]. Celem tego procesu jest pod wpływem czynników chemicznych lub wpływu wysokiej temperatury doprowadzić do rozkładu (tzw. denaturacji) pierwotnych nici kwasów nukleinowych do fragmentów a następnie w wyniku usunięcia wpływu czynnika umożliwić ich ponowne łączenie.

Zgodnie z treścią rozprawy doktorskiej dr. Marcina Radom [5], wykorzystując wyżej omówione procesy, chip DNA oraz metody kombinatoryczne, istnieje możliwość odtworzenia kolejności nukleotydów w badanej cząsteczce DNA. W wielkim uproszczeniu proces, ten polega na dokonaniu hybrydyzacji badanej cząsteczki DNA w celu pozyskania fragmentów DNA. Chip DNA stanowi urządzenie mające za zadanie rozpoznania poszczególnych fragmentów DNA i utworzenia ich cyfrowych reprezentacji. Na tym etapie

kończy się faza biochemiczna a zaczyna faza obliczeniowa. Od tego momentu dane uzyskane z chipu DNA stanowią dane wejściowe dla metod kombinatorycznych, których użycie jest niezbędne, aby odtworzyć kolejność nukleotydów w poddanej hybrydyzacji cząsteczce DNA. W tym przypadku dla problemu TSP reprezentacje poszczególnych fragmentów DNA stanowią dla komiwojażera miasta, a ilość błędów odzwierciedlenia początkowej nici DNA stanowi koszt uzyskanego rozwiązania.

1.3.2 Optymalizacja ruchów robota do produkcji mikrochipów

Produkcja mikrochipów z uwagi na pracę w małej skali oraz duży stopień skomplikowania układów to bardzo trudny dla człowieka proces. Postępujący rozwój technologiczny sprawił, że dzisiaj już mało ludzi ręcznie wytwarza w pełni samemu własne płytki drukowane wlotowując w nie już gotowe układy scalony i elementy elektroniczne by po wykonaniu wszystkich operacji, układ scalony działał zgodnie z wizją autora. Za sprawą postępującej robotyzacji dzisiaj mikrochipy produkuje się przemysłowo na skalę masową. Pomimo, że maszyny nie ulegają zmęczeniu i wykonują operacje wlotowywania elementów taniej, lepiej i szybciej niż człowiek, to nadal można jeszcze dokonać optymalizacji realizowanych przez nich operacji. Jednym z pomysłów by zmniejszyć ilość czasu potrzebnego na wlotowanie dużej ilości elementów na płytkę PCB była modyfikacja sekwencji ruchów robota zgodnie z modelem TSP, który to został zbadany w 2003 przez Ratnesh Kumar'a oraz Zhongui Luo na Uniwersytecie w Kentucky [6]. Zgodnie z wynikami ich pracy zastosowanie modelu TSP przy wlotowywaniu elementów na płytkę PCB w zależności od liczby elementów i rozmiaru płytki potrafiło skrócić czas wyrobu płytki o około 25% - 33%. Za sprawą rozwiązania problemu TSP ramię robota musiało pokonać najmniejszy możliwy dystans by wykonać wszystkie niezbędne operacje umieszczania elektronicznych komponentów na płycie drukowanej w jak najkrótszym czasie.

1.4 TSP a metody heurystyczne

Zaznajamiając się z trudnościami związane z zagadnieniem TSP, należałoby się zastanowić, czy ludzkość posiada narzędzia do rozwiązywania tak skomplikowanego problemu. Jednym z tych narzędzi są metody heurystyczne. Aby móc dobrze zrozumieć sposób w jaki rozwiązują problemy kombinatorycznych, należy najpierw zrozumieć samo pojęcie heurystyki oraz metaheurystyki a także sposobów podziału tych drugich.

Heurystyka (gr. *Heuriskō* – znajduję) zgodnie z definicją z encyklopedii PWN jest to umiejętność wykrywania nowych faktów i związków między faktami, zwłaszcza czynność formułowania hipotez (przeciwstawiana czynności uzasadniania) prowadząca do poznania nowych prawd naukowych [7]. W informatyce **heurystyką** nazywamy metodą rozwiązywania konkretnego problemu, która znajduje dobre rozwiązania przy akceptowalnych nakładach obliczeniowych, ale bez gwarancji, że znalezione rozwiązanie jest najlepsze z możliwych [8].

Metaheurystyka to ogólna metoda służąca za szkielet do konstrukcji heurystyki rozwiązującej dowolny problem, który można opisać za pomocą pewnych definiowanych przez tę metodę pojęć [9]. Metody takie nie służą do rozwiązywania konkretnych problemów, a jedynie podają sposób na utworzenie odpowiedniego algorytmu heurystycznego.

Cechy metaheurystyk [9]:

- opracowywanie strategii określających sposób przeszukiwania przestrzeni dopuszczalnych rozwiązań
- celem działania jest efektywne przeszukiwanie przestrzeni
 - znajdowanie dobrych rozwiązań w określonym regionie (**eksploatacja**)
 - przeglądanie możliwie najszerszego obszaru przestrzeni problemu (**eksploracja**)
- metody przybliżone i zazwyczaj niedeterministyczne
- stosują różne techniki: od prostego przeszukiwania lokalnego do skomplikowanych procesów ewolucyjnych

- wykorzystują mechanizmy zapobiegające utknięciu metody w ograniczonym obszarze przestrzeni problemu
- nie są specjalizowane do żadnego specyficznego problemu
- wykorzystują wiedzę o problemie i/lub doświadczenie zgromadzone podczas przeszukiwania przestrzeni

Podział metaheurystyk ze względu na [9]:

- Inspiracje
 - Inspirowane przyrodą tj. naśladujące zjawiska biologiczne i społeczne
 - algorytmy ewolucyjne
 - algorytmy mrówkowe
 - sztuczne systemy immunologiczne
 - optymalizacja rojem cząstek
 - ewolucja kulturalna
 - przeszukiwanie z użyciem zjawiska „tabu”
 - Inspirowane zjawiskami fizycznymi i chemicznymi
 - symulowane wyżarzanie
 - Inspirowane poza przyrodniczo
 - ILS metoda iterowana metodą lokalnych poszukiwań
 - VNS przeszukiwanie zmiennego sąsiedztwa
- Ilość znajdowanych przez nie rozwiązań
 - poszukujące wielu rozwiązań (populacyjne)
 - znajdujących pojedyncze rozwiązanie (niepopulacyjne)

Metaheurystyki inspirowane naturą zawsze mają na celu odwzorowanie w cyfrowym świecie działania metod, które zostały wypracowane przez naturę na drodze ewolucji oraz specjalizacji gatunkowej, konkretnych rodzajów flory i fauny. Do tej grupy zaliczane są również sposoby rozwiązywania problemów bazujące na analizie ruchów społecznych. Dzięki symulowaniu zachowań behawioralnych ludzi i zwierząt w środowisku komputerowym można użyć ich do rozwiązania postawionego problemu w kreatywny sposób wykorzystując wiedzę nie tylko jednostki, ale również grupy.

Metody bazujące na zjawiskach fizycznych i chemicznych mają na celu zasymulowanie w środowisku cyfrowym reguł oraz praw jakim podlega materia w świecie rzeczywistym by następnie zasymulować serię eksperymentów chemicznych i fizycznych na cyfrowej materii, która stanowi zbiór wszystkich dostępnych rozwiązań postawionego problemu.

Algorytmy inspirowane poza przyrodniczo bazują na połączeniu podejścia stochastycznego modyfikowania rozwiązania wraz z jego lokalną optymalizacją i weryfikowaniem czy zaindukowana modyfikacja pozwoliła na uzyskanie lepszego rozwiązania.

1.5 Motywacja

Podsumowując, całą problematykę zagadnienia TSP oraz opracowane już metody do jego rozwiązania wykorzystujące podejście heurystyczne autor pracy uznał go za interesujący obszar do prowadzenia badań. Badania omawianego problemu stanowią dla autora okazję do poznania już istniejących sposobów rozwiązywania problemu komiwojażera w oparciu o różne dokonania informatyczne i matematyczne poczynając od wykorzystania podstawowej wiedzy z teorii grafów i kombinatoryki aż po wykorzystanie programowania dynamicznego, algorytmów heurystycznych w tym algorytmów ewolucyjnych.

Autora również cechuje zamiłowanie do tworzenia oprogramowania i chęć ciągłego rozwoju, dlatego jako język programowania do realizacji swoich badań wybrał on język Python, który pozwala na programowanie funkcjonalne jak i obiektowe oraz posiada dostęp do szerokiego zakresu gotowych bibliotek pozwalających na wykorzystanie i porównanie gotowych już zaimplementowanych algorytmów heurystycznych i opracowanie do ich porównywania stosownej programistycznej architektury pomiarowej.

Pomimo, że przy rozwiązywaniu problemów optymalizacyjnych powinien być wybrany język, który jest uważany za najszybszy (obecnie C/C++), to autor zdecydował się wykorzystać język Python, z uwagi na jego wciąż rosnącą popularność oraz tym, że w roku 2021 zgodnie z raportem „The software quality company” [10] wyprzedził on język C i C++ jak również zostawił on uwielbianą przez szerzę fanów tworzących aplikacje Enterprise Javę w tyle.

2. Cel i założenia pracy

Celem pracy było zaznajomienie się z istniejącymi już pracami badawczymi opisującymi różne podejścia do rozwiązania omawianego problemu TSP oraz porównanie ich wydajności oraz optymalności uzyskanych z ich pomocą rozwiązań. Autor do zbadania różnych typów algorytmów zamierza do ich zbadania użyć następujących metryk:

- Czas wykonania algorytmu mierzony w sekundach
- Ilość zaalokowanej pamięci operacyjnej w bajtach zmierzonej z użyciem narzędzia tracemalloc [11]
- Średnie zużycie procesora, do którego pomiaru została wykorzystana biblioteka psutil [12]
- Koszt uzyskanego rozwiązania (suma odległości miast, które odwiedził komiwojażer)
- Czy uzyskane rozwiązanie jest optymalne porównując koszt i trasę oraz jej odwrotność wyznaczoną dla danego zestawu miast przez algorytm brutalnej siły.
- Błąd bezwzględny (różnica między kosztem uzyskanego rozwiązania a kosztem uzyskania rozwiązania optymalnego (uzyskany minimalny koszt dla optymalnego rozwiązania))
- Błąd względny ($[\text{błąd bezwzględny} / \text{koszt minimalny}] * 100 \%$)

Autor w swojej pracy zamierza opisać działanie oraz zbadać następujące algorytmy heurystyczne z wykorzystaniem powyżej przedstawionych metryk:

1. Algorytmy brutalnej siły
 - a. algorytm Bellmana-Helda-Karpa (programowanie dynamiczne)
2. Algorytmy zachłanne (prosta heurystyka)
 - a. Algorytm greedy search
3. Algorytmy heurystyczne
 - a. Algorytm A*
 - b. Algorytm symulowanego wyżarzania
 - c. Algorytm iteracyjnego lokalnego przeszukiwania

- d. Algorytm genetyczny
- e. Algorytm mrówkowy
- f. Algorytm roju cząstek

Do zebrania danych pomiarowych autor zamierza wykorzystać sprzęt o poniżej przedstawionych parametrach, których szczegółowa dokumentacja znajduje się załącznikach.

Tabela 2 Tabela prezentująca sprzęt wykorzystany do badań

Rodzaj sprzętu	Opis urządzenia
Laptop	Lenovo Y700-15ISK [13]
Procesor (CPU)	Procesor Intel® Core™ i7-6700HQ (pamięć cache 6 MB, nawet do 3,50 GHz) [14]
Pamięć operacyjna RAM	<ul style="list-style-type: none">• 2 x Kingston HyperX KHX2133C13S4<ul style="list-style-type: none">○ Taktowanie: 2133 MHz○ Opóźnienie: (cycle latency) CL13○ Napięcie: 1,2 V○ Rodzaj: DDR4 SODIMM
Dysk twardy, pamięć ROM	Samsung SSD 960 EVO 250GB [15] CT500MX500SSD1 500GB [16]
System operacyjny	Windows 10 wersja 21H1 (kompilacja 19043.1466)

3. Omówienie badanych algorytmów i bibliotek

W ramach badań autor wykorzystał implementacje poszczególnych algorytmów z istniejących już bibliotek powstałych do rozwiązywania problemów optymalizacyjnych z użyciem języka python. Każda z omówionych poniżej bibliotek stanowi narzędzie open source co sprawia, że każdy użytkownik pythona może je wykorzystać w ramach własnych projektów programistycznych.

3.1 Biblioteka „python-tsp”

Jednym z wykorzystanych przez autora narzędzi była biblioteka „python-tsp” utworzona przez Filipe Goulart i udostępniona w ramach licencji MIT [17]. Autor owego narzędzia zaimplementował 4 algorytmy w tym 2 znajdujące rozwiązanie dokładne oraz 2 wykorzystujące podejście heurystyczne. Do badań z tej biblioteki zostały wykorzystane wszystkie poniżej przedstawione algorytmy:

- algorytmy znajdujące zawsze najlepsze rozwiązanie
 - algorytm stosujący metodę siłową (ang. brute force)
 - algorytm Helda-Karpa oparty o programowanie dynamiczne
- algorytmy heurystyczne (nie dają gwarancji znalezienia najlepszego rozwiązania)
 - algorytm symulowanego wyżarzania
 - algorytm lokalnego przeszukiwania

3.1.1 Metoda siłowa

Metoda siłowa (ang. brute force) znana jest również w środowisku naukowym pod postacią wyszukiwania wyczerpującego (ang. exhaustive search) jest to sposób by znaleźć najlepsze rozwiązanie przez zweryfikowanie wszystkich możliwych. Metoda ta, pomimo że zapewnia zawsze rozwiązanie optymalne to jednak jest skuteczna wtedy i tylko wtedy, gdy przestrzeń dostępnych rozwiązań jest względnie mała. Natomiast w innych przypadkach metoda ta jest zbyt kosztowna w zasoby i nie pozwala ona w stosownym dla użytkownika czasie rozwiązać danego zagadnienia. W najbardziej optymistycznym przypadku, gdzie przestrzeń rozwiązań rośnie linowo jak np. podczas poszukiwania minimum w tablicy n - elementowej, czyli posiada złożoność obliczeniową $O(n)$.

W rozważanym problemie symetrycznego TSP przestrzeń rozwiązań rośnie wykładniczo dokładnie dla N miast metoda ta wymaga sprawdzenia $(N-1)! / 2$ dostępnych permutacji N liczb, co jest równoznaczne z tym, że wraz ze wzrostem N wykładniczo rośnie również czas wymagany do zrealizowania wszystkich operacji porównania. Poniżej został przedstawiony kod źródłowy wykorzystany do badań pochodzący z bibliotek „python-tsp”.

3.1.2 Algorytm Bellmana-Helda-Karpa

W 1962 roku uczeni Bellman, Held oraz Karp [18] zaproponowali nowatorskie rozwiązanie symetrycznego problemu komiwojażera z użyciem strategii, która znana jest pod nazwą programowania dynamicznego [19].

Idea programowania dynamicznego, opiera się o podzieleniu pierwotnego dużego problemu na wiele pod problemów w myśl zasady „dziel i zwyciężaj” (ang. divide and conquer) [20] oraz zapisanie ich rozwiązań w strukturze danych (zazwyczaj tablicy haszującej) zapewniającej dostęp do danego rozwiązania w czasie jednostkowym. Ten typ programowania posiada zastosowanie do rozwiązywania problemów, których optymalne rozwiązanie stanowi funkcję optymalnych rozwiązań pod problemów.

Algorytm Bellmana-Helda-Karpa wykorzystuje powyżej wyjaśnioną strategię, aby kawałek po kawałku przybliżać się do rozwiązania optymalnego by w końcu je osiągnąć. Zgodnie z danymi z elektronicznej encyklopedii algorytmów [21] złożoność czasowa przedstawionego sposobu wynosi $O(n^2 2^n)$ a złożoność pamięciowa $O(n 2^n)$. Poniżej zostało przedstawione działanie algorytmu oraz kod z biblioteki python-tsp wykorzystany podczas badań.

Graf liczący n wierzchołków (miast) ponumerowanych od 1, 2... n . Wierzchołek 1 niech będzie startowym oraz jako $d_{i,j}$ oznaczmy odległość między wierzchołkami o numerach i oraz j .

Przebieg algorytmu:

Jeśli $s = 1$, to $D(S, p) = d_{1,p}$

Jeśli $s > 1$, to $D(S, p) = \min_{x \in (S - \{p\})} (D(S - \{p\}, x) + d_{x,p})$

gdzie:

S – zbiór wierzchołków, przez które musi przebyć komiwojażer by dotrzeć do wierzchołka p

s – ilość wierzchołków znajdujących się w zbiorze S

p – numer wierzchołka początkowego oraz końcowego w poszukiwanym rozwiązaniu

3.1.3 Algorytm symulowanego wyżarzania

3.1.3.1 Metalurgia, inspiracją dla naukowców

Symulowane wyżarzanie (ang. simulated annealing) opiera się na odtworzeniu w świecie cyfrowym procesu metalurgicznego w którym zamiast metalu wykorzystujemy nasz rozpatrywany problem a właściwie zbiór wszystkich możliwych rozwiązań naszego zagadnienia [22]. Metal jak każda znana ludzkość materia składa się z atomów. Drobiny te w środowisku, w którym panuje wysoka temperatura są bardziej ruchliwe niż w temperaturze pokojowej. Wzmogłą ruchliwość elementów składowych metalu pod wpływem działania wysokiej temperatury obserwujemy jako żarzenie się metalu. Powolny spadek jego temperatury tj. stygnięcie stopu metalu nazywamy wyżarzaniem. Celem takiej obróbki jest przybliżenie stanu materiału do warunków równowagi termodynamicznej, czyli takich warunków, w których jego właściwości stają się niezmiennie w czasie [23]. Żarzący się stop metalu jest o wiele bardziej elastyczny i podatny na zmianę jego pierwotnego kształtu oraz właściwości, natomiast pod wpływem obniżania się jego temperatury ruchliwość jego atomów oraz jego kowalność stopniowo zanika. Po osiągnięciu przez nasz materiał stabilnej temperatury, można zauważyć, że pierwotna postać metalu uległa

modyfikacji, a w przypadku, gdy środowiskiem dla naszego metalu była forma odlewnicza następuje utworzenie stabilnej struktury, o zadanym kształcie.

Powyższe zjawisko stało się inspiracją dla uczonych do opracowania rodzaju algorytmu heurystycznego mającego za zadanie przeszukiwania przestrzeni alternatywnych rozwiązań problemu w celu wyszukania rozwiązań najlepszych. [24].

3.1.3.2 Ogólny zarys działania metaheurystyki

W celu zaimplementowania owego rozwiązania należy na początku zdefiniować, zestaw własności i warunków początkowych, których wartość jest istotna dla prawidłowego przebiegu naszej symulacji wyżarzania do rozwiązania problemu kombinatorycznego. Opis tych własności został przedstawiony w punktach poniżej.

Czynności początkowe wymagane do zastosowania symulowanego wyżarzania [25]:

1. Ustalić początkową wartość temperatury T (temperaturę żarzenia się problemu, jego największej elastyczności)
2. Zdefiniować sposób obniżania temperatury (zazwyczaj przybiera on formę mnożenia aktualnej temperatury przez współczynnik z przedziału $[0.8; 0.99]$)
3. Zdefiniować liczbę prób przeprowadzanych w ramach jednej epoki (przy tej samej temperaturze)
4. Ustalić sposób wyboru nowego rozwiązania w ramach pojedynczej próby. Nowe rozwiązanie powinno być zbliżone do obecnego. Przy wyznaczaniu nowego rozwiązania można uwzględnić temperaturę – im wyższa tym bardziej nowe i obecne rozwiązanie może się od siebie różnić
5. Określić warunek zatrzymania symulacji, może on przyjąć postać osiągnięcia pewnej liczby epok lub wykrycia odpowiednio małej zmiany rozwiązania w zakończonych epokach

Kroki działania algorytmu [25]:

1. Wylosuj rozwiązanie początkowe X
2. Wybierz losowe rozwiązanie X' bliskie rozwiązaniu X
3. Jeśli rozwiązanie X' jest lepsze niż X przyjmij je. W przeciwnym razie znajdź prawdopodobieństwo przyjęcia nowe rozwiązanie zgodnie ze wzorem $e^{\frac{f(X)-f(X')}{T}}$. Następnie wybierz liczbę z zakresu $[0,1]$ i jeśli jest mniejsza niż obliczone

prawdopodobieństwo, należy przyjąć nowe rozwiązanie (pomimo faktu, że jest gorsze)

4. Jeśli nie ukończyłeś jeszcze liczby prób w danej epoce wróć do kroku 2.
5. Zmniejsz temperaturę
6. Jeśli warunek zatrzymania nie został jeszcze osiągnięty, wróć do kroku 2

3.1.4 Algorytm przeszukiwania lokalnego

LS (ang. local search) stanowi jedną z bardzo prostych a zarazem niezwykle użytecznych metaheurystyk [26]. Podstawowym założeniem tego algorytmu jest modyfikowanie obecnie wybranego rozwiązania tworząc nowe dzięki czemu nowa propozycja rozwiązania jest ulepszoną wersją starego i przeszukujemy N wymiarową przestrzeń naszych rozwiązań w sposób ciągły w przeciwieństwie gdybyśmy za każdym razem losowo generowali kolejne rozwiązanie. Po każdym ulepszeniu rozwiązania następuje jego ocena, aby obecne zastąpić lepszym. Wykonanie algorytmu kończy się, gdy zostanie jeden ze zdefiniowanych warunków zatrzymania zostanie spełniony. Jego działanie jest podatne na wpadanie w minima lokalne funkcji celu. Najlepiej działanie LS zobrazuje poniższa lista kroków.

Opis algorytmu LS

1. Wygeneruj rozwiązanie startowe - s_0
2. Zmodyfikuj rozwiązanie s_0 tworząc rozwiązanie s
3. Zweryfikuj czy warunek stopu został osiągnięty, jeśli nie przejdź do kroku 4
4. Oceń rozwiązanie s i jeśli jest lepsze niż s_0 zastąp je rozwiązaniem s
5. Zweryfikuj czy warunek stopu został osiągnięty, jeśli nie przejdź do kroku 2

Najczęściej jako warunek zatrzymania stosuje się osiągnięcie przez algorytm zadanej liczby epok, lub wykrycie, że nowe lepsze rozwiązanie nie zostało znalezione od k epok. Jedną z trudności przy implementowaniu tego algorytmu jest wybranie algorytmu do przekształcenia obecnego rozwiązania w nowe lepsze. Ten element jest tak bardzo istotny, ponieważ jego działanie powinno opierać się o wszystkie poniżej zdefiniowane cechy.

Cechy algorytmu ulepszającego/perturbującego obecne rozwiązanie:

- Nowe rozwiązanie musi być wystarczająco różne od starego by uniknąć wpadania w te same minima lokalne
- Nowe rozwiązanie musi być wystarczająco podobne do starego by uniknąć, całkowitego losowego przeszukiwania przestrzeni rozwiązań

3.1.5 Cechy biblioteki python-tsp

Samo algorytmy heurystyczne tak naprawdę nie są szczegółowo opisanymi algorytmami a stanowią jedynie ogólną koncepcję. Dzięki takiemu luźnemu podejściu wiele elementów może zostać różnie zaimplementowanych przez autora i oddanych do użytku przez użytkownika. Podobnie wygląda kwestia w przypadku biblioteki python-tsp [17], której autor zaimplementował do użytku użytkowników kilka poniżej opisanych usprawnień.

1. Możliwość zalogowania całego przebiegu algorytmu do pliku
2. Warunkiem stopu może być również czas wykonania algorytmu podany w sekundach
3. Algorytm ulegnie zatrzymaniu, jeśli w ciągu ostatnich 3 epok znalezione rozwiązanie nie uległo poprawie
4. Autor w bibliotece oferuje użytkownikom wykorzystania kilka algorytmów umożliwiających wygenerowanie nowego rozwiązania na podstawie obecnego co nazwał schematami perturbacji (ang. `pertrubation_scheme`):
 - a. PS1 – zamiana dwóch sąsiadujących ze sobą elementów ($n-1$ zamian)
 - b. PS2 – zamiana dowolnych dwóch elementów ($n*(n-1) / 2$ zamian)
 - c. PS3 – przeniesienie pojedynczego elementu ($n*(n-1)$ zamian)
 - d. PS4 – przeniesienie podciągu
 - e. PS5 – odwrócenie podciągu
 - f. PS6 – odwrócenie i przeniesienie podciągu
 - g. two-opt-gen – (domyślny) wykorzystanie algorytmu 2-opt [27]

3.2 Algorytm A*

Algorytm A* to jeden z algorytmów heurystycznych do znajdowania najkrótszej ścieżki w grafie. Jest to algorytm zupełny co oznacza, że zawsze znajdzie najlepsze rozwiązanie problemu, jeśli funkcja heurystyczna została poprawnie zdefiniowana. W tej metodzie istnieje model pamięciowy, który zapewnia odwiedzenie możliwości odwiedzenia każdego wierzchołka grafu. A* stanowi przykład metody „najpierw najlepszy” oraz działa najlepiej, gdy przestrzeń przeszukiwać jest w postaci drzewa lub grafu.

3.2.1 Działanie algorytmu

A* opiera swoje działanie na minimalizacji funkcji celu $f(x)$, zdefiniowanej jako suma kosztu funkcji $g(x)$ oraz funkcji heurystycznej $h(x)$ [28].

$$f(x) = g(x) + h(x)$$

Co każdy krok algorytm A* wydłuża utworzoną ścieżkę o kolejny wierzchołek grafu, wybierając ze zbioru jeszcze nie odwiedzonych wierzchołków ten z minimalną wartością funkcji f .

Funkcja $g(x)$ wyznacza rzeczywisty koszt dotarcia do punktu x (suma wag krawędzi należących już do ścieżki oraz wagi krawędzi łączącej aktualny węzeł z x).

Funkcja $h(x)$ stanowi funkcję heurystyczną, która jest odpowiedzialna za oszacowanie (zawsze optymistyczne) koszt dotarcia od punktu x do wierzchołka docelowego.

Prawidłowa funkcja heurystyczna musi spełniać dwa warunki [29]:

- Warunek dopuszczalności: $g(x) + h(x) \leq g(x_t)$
- Warunek monotoniczności: $g(x_j) + h(x_j) \geq g(x_i) + h(x_1)$

gdzie: $j > i$, a x_t oznacza punkt końcowy

Warunek dopuszczalności determinuje nadmierny optymizm. Funkcja heurystyczna, jest odpowiedzialna za niedoszacowanie, gdy minimalizujemy koszt uzyskania naszego rozwiązania, a przeszacowuje, gdy maksymalizujemy zysk.

Warunek monotoniczności ma za zadanie sprawić, że im bliżsi jesteśmy rozwiązania tym oszacowanie musi być coraz mniej optymistyczne.

Kluczową funkcją algorytmu A* jest możliwość nawrotów. Jest to możliwe dzięki oszacowaniu z wykorzystaniem poprawnie zdefiniowanej funkcji heurystycznej. Algorytm dzięki niej potrafi diametralnie zmienić jej strukturę ścieżki komiwojażera by z każdym kolejnym przeskokiem zbliżać się do rozwiązania optymalnego.

3.2.2 Opis algorytmu A*

Algorytm A* trudno jest opisać w postaci listy, kroków, dlatego, by lepiej wyjaśnić jego działanie autor pracy przedstawia poniżej jego reprezentację w pseudokodzie

```
function A*(start,goal)
    closedset := the empty set           % Zbiór wierzchołków przejranych.
    openset := set containing the initial node % Zbiór wierzchołków nieodwiedzonych, sąsiadujących z
    odwiedzionymi.
    g_score[start] := 0                  % Długość optymalnej trasy.
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from,goal)
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)
            tentative_is_better := false
            if y not in openset
                add y to openset
                h_score[y] := heuristic_estimate_of_distance_to_goal_from(y)
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            if tentative_is_better = true
                came_from[y] := x
                g_score[y] := tentative_g_score
                f_score[y] := g_score[y] + h_score[y] % Przewidywany dystans od startu do celu przez y.
    return failure

function reconstruct_path(came_from,current_node)
    if came_from[current_node] is set
        p = reconstruct_path(came_from,came_from[current_node])
        return (p + current_node)
    else
        return the empty path
```

Rysunek 2 Algorytm A* w pseudokodzie [28]

3.3 Algorytm najbliższego sąsiada (greedy search)

Algorytm NN (ang. nearest neighbour) należy do grupy algorytmów zachłannych [30]. Wykorzystywana przez niego metaheurystyka jest bardzo prosta, komiwojażer wybiera następne miasto do odwiedzenia wybierając to do którego ma najbliżej z obecnego położenia. Gdy komiwojażer, gdy odwiedzi już wszystkie miasta, algorytm kończy działanie z wypracowaną przez niego ścieżką. By lepiej zrozumieć, jak działa ten algorytm poniżej zamieszczono jego opis w postaci listy kroków.

Opis algorytmu NN:

1. Wybierz wierzchołek startowy, dodaj go do listy odwiedzonych oraz ustaw koszt=0
2. Dodaj wszystkie nieodwiedzone wierzchołki do zbioru
3. Wyznacz z nieodwiedzonych wierzchołków ten znajdujący się najbliżej wierzchołka znajdującego się na końcu listy odwiedzonych
4. Powiększ koszt o dystans wyznaczony w kroku 3, dodaj wybrany wierzchołek do listy odwiedzonych oraz usuń go ze zbioru nieodwiedzonych
5. Jeśli zbiór wierzchołków nieodwiedzonych nie jest pusty przejdź do kroku 3
6. Do kosztu dodaj dystans pomiędzy ostatnim wierzchołkiem z listy odwiedzonych a wierzchołkiem startowym

Pomimo że wydawać by się mogło, że z jego użyciem komiwojażer postępuje w sposób logiczny i przemyślany to niestety ta strategia nie daje gwarancji znalezienia rozwiązania optymalnego. Rozwiązania przez nią wypracowane są średnio o 25% gorsze niż rozwiązanie optymalne [31]. Jedną z wad tego algorytmu jest również to, że istnieje taki zestaw danych wejściowych, który zapewnia, że algorytm NN zapewni zawsze najgorsze możliwe rozwiązanie.

3.4 Algorytm genetyczny

Algorytm genetyczny (ang. genetic algorithm) stanowi rodzaj heurystyki, mającej za zadanie przeszukiwania przestrzeni alternatywnych rozwiązań w celu znalezienia najlepszego. Powstał on na podstawie obserwacji, że różne organizmy na Ziemi konkurując ze sobą o zasoby naturalne i podlegając ciągłym zmianom genetycznym podczas rozmnażania uczestniczą w ciągłej ewolucji, podobnie jak ewolucji podlegają różne wojenne techniki ataku i obrony czy też techniki wykonywania skoków narciarskich [32]. Za twórcę GA uznaje się Johna Henry Hollanda, który zainspirowany zjawiskiem ewolucji biologicznej doszedł do wniosku, że procesy ewolucyjne mogą zostać wykorzystane do rozwiązania różnych problemów [33]. Algorytm w tym celu posługuje się zdefiniowaniem populacji początkowej dostępnych rozwiązań, po czym podawaniu jej szeregu procesów ewolucyjnych takich jak selekcja osobników, krzyżowanie genetyczne oraz mutacja genetyczna [34]. Upraszczając, aby przy użyciu komputera rozwiązać skomplikowany problem używając metody ewolucyjnej, należy zamiast weryfikować olbrzymia liczbę możliwych rozwiązań danego problemu, należy stworzyć niewielki zbiór (populację) tych rozwiązań, a następnie dokonywać ich modyfikacji (mutacji), wymieniać ich części (krzyżowanie), oceniać, oraz przede wszystkim powielać (rozmnażać) te, które są najlepsze w populacji.

3.4.1 Terminologia

W celu przejścia do właściwego schematu działania AG należy najpierw wyjaśnić terminologię, którą autor pracy będzie się posługiwał w ramach dalszego rozwijania owego tematu.

Terminologia AG:

1. allele – wartości, warianty genów
2. chromosom – (inaczej osobnik, genotyp, struktura, łańcuch lub ciąg kodowy) zakodowane zmienne/ parametry problemu
3. fenotyp – zdekodowane zmienne/parametry zadania, rozwiązanie
4. gen – (inaczej cecha, znak, dekodery) zakodowana wartość pojedynczej zmiennej/parametru
5. generacja – populacja w wymiarze czasowym (w kolejnych iteracjach)

6. krzyżowanie – (inaczej rekombinacja) operator genetyczny dwu lub wieloargumentowy, łączący cechy osobników rodzicielskich w osobnikach potomnych
7. locus – pozycja genu w chromosomie
8. mutacja – operator genetyczny jednoargumentowy, wprowadzający perturbację chromosomu
9. napór selekcyjny – wymagania środowiska względem osobników potomnych
10. populacja – zbiór chromosomów przetwarzanych w procesie ewolucyjnym
11. przystosowanie – ocena chromosomu, miara jak dany chromosom/rozwiązanie rozwiązuje zadany problem
12. pula rodzicielska – tymczasowa populacja chromosomów utworzona w wyniku selekcji
13. selekcja – (inaczej reprodukcja) proces powielania chromosomów w stosunku zależnym od ich przystosowania

3.4.2 Schemat działania AG

W celu lepszego zrozumienia zasadę działania algorytmu genetycznego dla problemu TSP autor pracy najpierw przedstawi ogólny zarys jego działania w postaci poniższej listy kroków.

Schemat działania AG [35]:

1. Zdefiniowanie zmiennych i ustalenie ich reprezentację
 - a. zdefiniuj dziedzinę problemu, przestrzeń dopuszczalnych rozwiązań mogących powstać w ramach działania AG
 - b. zdefiniuj sposób kodowania chromosomów (liczby genów z jakich się składają oraz ilość alleli i zakres wartości przez nich reprezentowany)
2. Ustalenie parametrów algorytmu
 - a. liczebność populacji
 - b. operator krzyżowania
 - c. prawdopodobieństwo krzyżowania
 - d. operator mutacji
 - e. prawdopodobieństwo mutacji

- f. metodę selekcji
 - i. selekcja z użyciem koła ruletki (ang. roulette wheel)
 - ii. selekcja rankingowa (ang. ranked selection)
- g. warunek zatrzymania
 - i. liczba generacji
 - ii. Wykrycie w populacji osobnika o docelowej wartości funkcji przystosowania
 - iii. Zadany procent populacji osiągnął lub przekroczył zadaną wartość funkcji przystosowania
 - iv. Wykrycie braku poprawy przystosowania osobników przez n ostatnich generacji
- 3. Wygenerowanie populacji początkowej $P(0)$
- 4. Ocena przystosowania osobników w populacji $P(0)$
- 5. Dla $t = 1, 2, \dots$ powtarzaj do spełnienia warunku zatrzymania
 - a. Dokonanie reprodukcji osobników
 - i. wybranie z populacji $P(t)$ do $P(t-1)$ osobników z wykorzystaniem wybranej metody selekcji
 - ii. Zmodyfikuj $P(t)$ używając operatorów genetycznych
 - 1. połącz losowo osobniki w pary tworząc pulę rodzicielską
 - 2. dla każdej pary wylosuj liczbę z zakresu 0-1 określając prawdopodobieństwo ich skrzyżowania
 - 3. dokonaj krzyżowania par których prawdopodobieństwo krzyżowania jest większe niż parametr AG o tej samej nazwie
 - 4. zastąp osobniki rodzicielskie ich osobnikami potomnymi
 - 5. dla każdego osobnika wylosuj liczbę z zakresu 0-1 określając prawdopodobieństwo jego mutacji
 - 6. dokonaj mutacji osobnika których prawdopodobieństwo mutacji jest większe niż parametr AG o tej samej nazwie
 - iii. Oceń populację $P(t)$

3.4.3 Operatory genetyczne oraz kodowanie dla TSP

Zagadnienie TSP ze względu na swoją problematykę wymaga specjalnego podejścia przy wykorzystywaniu algorytmu ewolucyjnego do jego rozwiązania. Pierwszym problemem

jest uzyskanie odpowiedzi na pytania jak zapisać „genetycznie” potencjalne rozwiązanie problemu, czyli jak zakodować osobnika?

Najczęściej stosowane są dwa sposoby kodowania (reprezentowania osobnika) [36]:

- reprezentacja ścieżkowa
- reprezentacja kolejnościowa

W reprezentacji ścieżkowej, inaczej zwanym kodowaniem permutacyjnym, w osobniku zapisane są po prostu numery miast w kolejności ich odwiedzenia. Problemem tego typu kodowania jest fakt, że dwa pozorne różnie zakodowane osobniki mogą reprezentować dokładnie to samo rozwiązanie.

Przykład dla TSP z 7 miastami:

- Osobnik A: [1, 4, 3, 6, 7, 2, 5]
- Osobnik B: [6, 7, 2, 5, 1, 4, 3] – osobnik A którego geny/allele zostały podane operacji przesunięcia o 4 prawo

Bazując na powyższym przykładzie prawdziwym jest stwierdzenie, że kodowanie kolejnościowe dla problemu TSP z N miastami dopuszcza do powstania (N-1) różnych osobników, które reprezentują to samo rozwiązanie problemu TSP, co dodatkowo rozszerza nam przestrzeń naszych możliwych rozwiązań co stanowi dodatkową trudność dla GA by znaleźć dość dobre rozwiązanie dla TSP.

Reprezentacja ścieżkowa wymaga użycia specjalnych operatorów genetycznych. Zastosowanie dla problemu TSP zwykłego krzyżowania jednopunktowego między dopuszczalnymi osobnikami powoduje, że w rezultacie otrzymujemy dwa osobniki potomne niedopuszczalne, ponieważ osobniki potomne mogą zawierać ten sam zestaw genów, czyli powtarzające się miasta, czyli wygenerowane rozwiązania nie spełniają założeń problemu TSP.

Przykład krzyżowania jednopunktowego dla dwóch osobników i problemu TSP dla 7 miast:

- Rodzic A [1 4 3 6 | 7 2 5] – brak duplikacji genów osobnik dopuszczalny
- Rodzic B [1 2 3 4 | 5 6 7] – brak duplikacji genów osobnik dopuszczalny
- Potomek AB [1 4 3 6 5 6 7] – duplikacja genów, osobnik niedopuszczalny
- potomek BA [1 2 3 4 7 2 5] – duplikacja genów, osobnik niedopuszczalny

W celu uniknięcia powyżej opisanego problemu dla notacji ścieżkowej zostały opracowane specjalne operatory krzyżowania takie jak: [37]

1. PMX - Partially Mapped Crossover [38]
2. OX - Order Crossover [39]
3. EX – Edge Crossover [40]
4. SXX – Subtour Exchanged Crossover [41]
5. PX – Partition Crossover [42]

W reprezentacji porządkowej (ang. ordinal representation) stanowi inne podejście kodowania trasy. Osobnik reprezentuje sobą kolejność w jakiej z pewnej początkowej listy są wybierane miast tworzące trasę. Kolejne liczby (geny/allele) określają, które z pozostałych na liście miast należy wziąć jako następne na trasie, np.: [37]

Przykład kodowania porządkowego dla TSP dla 9 miast:

- lista miast: [1 2 3 4 5 6 7 8 9]
- osobnik: [1 1 2 1 4 1 3 1 1]
- dają w efekcie trasę: [1 2 4 3 8 5 9 6 7]

Kodowanie odbywa się zgodnie z następujących schematem, czyli bierzemy z listy po kolei (zgodnie z informacjami zawartymi w osobniku): miasto nr 1, następnie pierwsze z pozostałych na liście, czyli miasto nr 2. Kolejnym miastem jest drugie z pozostałych po wykorzystanych już miastach 1 i 2, a w tym jest miasto nr 4 itd [37].

Pomimo bardziej skomplikowanego podejścia i wymaganych pewnych dodatkowych operacji związanych z dekodowaniem osobnika, to kodowanie porządkowe rekompensuje problemy związane z krzyżowaniem i mutacją. Jest to możliwe, ponieważ na i -tej pozycji jest liczba z przedziału od 1 do $n-i+1$ (gdzie n to liczba wszystkich miast). Z tego powodu wymiana materiału genetycznego z użyciem standardowego x -punktowego operatora krzyżowania zawsze daje dopuszczalne osobniki potomne [37] [43].

W przypadku reprezentacji porządkowej dla problemu TSP należy również zastosować jeden ze specjalnie przeznaczonych do tego problemu operatorów mutacji.

Operatory mutacji dla reprezentacji porządkowej [37] [43]:

- inversion mutation – wybieranie losowego podciągu miast i odwracanie ich kolejności

- insertion mutation – przestawianie losowo wybranych miast na inną pozycję
- displacement mutation – zamienianie w wybranym losowo podciągu miast pierwszego miasta z ostatnim
- transposition (exchange) mutation – zamienianie dwóch losowo wybranych miast

3.4.4 AG porównanie biblioteki mlrose i scikit-opt

Algorytm genetyczny stanowi ogólną koncepcję z jakich elementów i procesów składa się symulacja ewolucji w świecie cyfrowym. Zazwyczaj, gdy opis danego podejścia jest bardzo ogólny wtedy powstaje wiele jego konkretnych implementacji. Tak właśnie jest w przypadku implementacji GA_TSP zapewnionej przez bibliotekę mlrose utworzoną przez Genevieve Hayesa do uczenia maszynowego, losowej optymalizacji oraz przeszukiwania [44] a także implementacji GA_TSP pochodzącej z biblioteki scikit-opt [45].

Implementacja AG_TSP w bibliotece mlrose nie pozwala na wybranie operatorów selekcji, krzyżowania oraz mutacji a także na ustalenia prawdopodobieństwa krzyżowania co jest równoznaczne z prawdopodobieństwem krzyżowania równym 100 % dla wszystkich osobników wyselekcjonowanych do rozmnażania się. W bibliotece mlrose dla problemu TSP jest używana selekcja metodą koła ruletki a krzyżowanie zostało zaimplementowane przy użyciu metody OX Order Crossover tworząc zawsze jednego osobnika potomnego zamiast dwóch (jak to zazwyczaj ma miejsce w AG) oraz korzysta z ustalania pojedynczego locus (punktu przecięcia).

Działanie OX w bibliotece mlrose dla locus = 3:

- parent A: [1 4 3 6 | 7 2 5]
- parent B: [6 7 2 5 1 4 3]
- potomek AB part 1: [1 4 3 6 0 0 0]
- nieodwiedzone wierzchołki porównując potomka AB z parentem B: [7 2 5]
- potomek AB part 2: [1 4 3 6 7 2 5]

```
570         # Reproduce parents
571         if self.length > 1:
572             _n = np.random.randint(self.length - 1)
573             child = np.array([0]*self.length)
574             child[0:_n+1] = parent_1[0:_n+1]
575             child[_n+1:] = parent_2[_n+1:]
576         elif np.random.randint(2) == 0:
577             child = np.copy(parent_1)
578         else:
579             child = np.copy(parent_2)
580     ---
```

Rysunek 3 Fragment kodu algorytmu genetycznego w bibliotece mlrose realizujący operację krzyżowania

W bibliotece mlrose został zastosowany operator mutacji typu wstawiającego losowo n losowych elementów. W poniżej przedstawionym kodzie w polu `max_val` znajduje się liczba miast dla obecnie rozpatrywanego problemu komiwojażera.

```
581         # Mutate child
582         rand = np.random.uniform(size=self.length)
583         mutate = np.where(rand < mutation_prob)[0]
584
585         if self.max_val == 2:
586             for i in mutate:
587                 child[i] = np.abs(child[i] - 1)
588
589         else:
590             for i in mutate:
591                 vals = list(np.arange(self.max_val))
592                 vals.remove(child[i])
593                 child[i] = vals[np.random.randint(0, self.max_val-1)]
594
595         return child
```

Rysunek 4 Fragment kodu realizujący mutację wstawiającą w AG z biblioteki mlrose

W przypadku implementacji scikit-opt również posiadamy implementację GA specjalnie przygotowaną do rozwiązywania problemu TSP. Podobnie jak w przypadku biblioteki mlrose tu również nie mamy możliwości ustawienia prawdopodobieństwa krzyżowania, gdy korzystamy z dedykowanej implementacji klasy do problemu TSP. Główną pozytywną różnicą między obiema bibliotekami jest możliwość wyboru operatora do selekcji, mutacji oraz krzyżowania z zaimplementowanych już przez autora algorytmów wymienionych poniżej.

Dostępne algorytmy selekcji w bibliotece sciki-opt:

- selekcja turniejowa – wykorzystuje do obliczeń pętlę
- selekcja turniejowa szybka – wykorzystuje do obliczeń operacje wektorowe
- selekcja z użyciem koła ruletki v1 – selekcja zapewniająca, że najgorzej przystosowany osobnik nie ma szansy zostać wybrany
- selekcja z użycie koła ruletki v2 – selekcja zapewniająca, że najgorzej przystosowany osobnik ma szansę zostać wybrany

Dostępne algorytmy krzyżowania w bibliotece scikit-opt:

- krzyżowanie 1-genowe
- krzyżowanie 2-genowe
- krzyżowanie 2-genowe binarne
- krzyżowanie 2-genowe z prawdopodobieństwem
- krzyżowanie typu PMX

Dostępne operatory mutacji w bibliotece scikit-opt:

- mutacja TSP_1 – każdy gen mutuje z każdym
- zamiana 2 genów
- mutacja typu 2-opt [46]
- transpozycja
- odwracanie kolejności wszystkich genów
- mutacja wykonująca negację bitową
- zamiana n genów

3.5 Algorytm mrówkowy

Algorytmy mrówkowe stanowią podzbiór algorytmów metaheurystycznych. Za jego twórcę uznaje się Marco Dorigo, który w 1992 na podstawie zachowań mrówek jako jednostek i działań kolonii mrowiska jako całości opracował ten stosunkowo nowy gatunek algorytmów optymalizacyjnych [47]. Typowym problemem dla tego gatunku owadów jest znalezienie pożywienia i przetransportowanie go do mrowiska. Badania wykazały, że pojedyncza mrówka ma bardzo ograniczoną percepcję i są zdolne tylko do przejawiania

prostych zachowań. Jednym z nich jest umiejętność wydzielania substancji chemicznych zwanych feromonami. Pełnią one funkcję tymczasowo trwałych znaczników, które stanowią dla mrówek mechanizm komunikacji, którego z czasem poziom intensywności maleje, ponieważ feromony odparowują z czasem. Dzięki temu mechanizmowi kolonie mrówek posiadają umiejętność do samoorganizacji, czyli potrafią z jego użyciem odnaleźć najkrótszą ścieżkę od źródła pożywienia do mrowiska.

3.5.1 Opis algorytmu mrówkowego

ACO stanowi dość nowy rodzaj algorytmu optymalizacyjnego, co sprawia, że jego definicja jest jeszcze dość płynna przez co trudno jest ocenić jaki algorytm zaliczamy do algorytmów mrówkowych, ale generalnie środowisko naukowe przyjęło, że algorytmy ACO posiadają poniższe 4 wspólne cechy.

Cechy algorytmów mrówkowych [48]:

1. optymalizacja realizowana jest w sposób iteracyjny przez pewną liczbę agentów
2. agenci komunikują się poprzez informację zostawianą w otoczeniu
3. agenci losowo eksplorują przestrzeń, faworyzując rozwiązania wskazywane im przez informację z otoczenia
4. agenci mogą posiadać własną pamięć, ale raczej nie posiadają rozbudowanej inteligencji

W celu zastosowania ACO do rozwiązania danego problemu należy postępować zgodnie z poniższą listą kroków.

Schemat działania ACO [48]:

1. Utwórz reprezentację problemu w postaci zadania reprezentowanego na grafie
2. Ustal początkową liczbę feromonów na krawędziach grafu
3. Ustal zestaw dodatkowych reguł za pomocą, których agenci będą poruszać się po grafie
 - a. zapamiętanie tablicy tabu już odwiedzonych wierzchołków
 - b. wybór pomocniczej heurystyki
4. Ustal w jaki sposób ruch agentów będzie wpływać na poziom feromonów na krawędziach grafu oraz jak agenci będą korzystać z już położonych feromonów

5. Iteracyjnie wprowadzaj nowe grupy agentów (po usunięciu starej grupy) i pozwalaj im przemieszczać się pomiędzy wierzchołkami grafu zgodnie z ustalonymi regułami
6. Po każdej iteracji zapamiętaj najlepsze dotychczas znalezione rozwiązanie

3.5.2 ACO a problem TSP

Algorytm mrówkowy został stworzony do rozwiązywania problemów kombinatorycznych a problem symetrycznego TSP stanowi jeden z ich rodzajów. Zanim algorytm mrówkowy zostanie wykorzystany dla problemu komiwojażera jego implementacja musi spełnić serię poniższych założeń.

Założenia ACO dla problemu TSP [48]:

1. Na każdej krawędzi grafu na początku należy nałożyć pewną stałą początkową ilość feromonów
2. Każda mrówka musi pamiętać odwiedzone miasta i już ich ponownie nie odwiedzać
3. Agenci zaczynają swój marsz w losowo wybranych miastach
4. Agenci poruszają się po grafie wybierając z użyciem metody ruletki następny wierzchołek, do którego przejdą kierując się ilością feromonów na krawędziach prowadzących do jeszcze nie odwiedzonych przez agenta wierzchołków
5. Aktualizacja ilości feromonów następuje po przejściu przez każdą mrówkę ścieżki Hamiltona. Na krawędzi ilość feromonu jest mnożona przez ustalony na początku czynnik z zakresu $[0,1]$ a następnie do odwiedzonych miast przez mrówki są dodawane wartości odwrotnie proporcjonalne do długości ich rozwiązań
6. Pamięć mrówek jest resetowana po aktualizacji przez nich ilości feromonu zanim zaczną ponowną eksplorację grafu

Jak każdy algorytm, algorytm mrówkowy również wymaga zdefiniowania pewnych parametrów przed jego użyciem, których opis znajduje się poniżej:

Parametry ACO:

- rozmiar populacji mrówek
- ρ - szybkość ulatniania się feromonu
- α – waga feromonu podczas wyboru ścieżki przez mrówkę

- beta - waga heurystyki podczas wyboru ścieżki
- ilość iteracji

Pomimo że powyższy opis algorytmu jest prosty i klarowny autor by uniknąć błędów związanych z jego implementacją postanowił skorzystać z gotowej implementacji oferowanej przez bibliotekę scikit-opt [49].

3.6 Algorytm optymalizacyjny z wykorzystaniem roju

Jednym z kolejnych algorytmów metaheurystycznych inspirowanych naturą jest PSO (ang. Particle Swarm Optimization) tak jak wcześniej opisane on również został opracowany z myślą rozwiązywania problemów optymalizacyjnych. Rozwiązanie to zostało zaproponowane w 1995 r. przez Kennedy’ego i Eberharda, którzy zostali zainspirowanie zachowaniem społecznym ptaków i ławic ryb [50].

Najlepiej działanie algorytmu wytłumaczyć bazując na przykładzie grupy ptaków poszukujących pożywienia w rozległej dolinie. Jedzenie znajduje się tylko w jednym miejscu w tej dolinie. Algorytm PSO stanowi model matematyczny, który stara się tłumaczyć jak to się dzieje, że żaden z ptaków nie wie, gdzie znajduje się jedzenie, ale wszystkie są świadome jak daleko od niego się znajdują [51].

Reasumując PSO opiera się na utworzeniu populacji agentów, którzy eksplorują przestrzeń alternatywnych rozwiązań naszego problemu w skończonej liczbie iteracji, którzy podczas przemieszczania się bazują na wiedzy własnej oraz swoich sąsiadów. Zachowanie to pozwala na symulację adaptacji roju do środowiska. Poruszające się cząstki poszukują najlepszego położenia w środowisku bazując na wiedzy swoich najbliższych sąsiadów [52].

3.6.1 Schemat działania algorytmu PSO

Jak to zostało wcześniej objaśnione PSO stanowi rodzaj algorytmu, gdzie za znalezienie „optimum” w naszym środowisku wykorzystujemy populację cząstek/agentów.

W celu opisanie dokładnego zachowania się cząstek autor pracy posłuży się poniższą listą kroków.

Opis działania algorytmu PSO do znajdowania ekstremum funkcji [51]:

1. Zdefiniuj funkcję, której cząstki będą poszukiwały ekstremum
2. Ustal parametry algorytmu lub zezwól na ich automatyczne ustawianie
 - a. maksymalna liczba iteracji
 - b. rozmiar populacji cząstek - N
 - c. alpha – wartość z zakresu R^+ , parametr definiujący poziom zaufania cząstki do własnej wiedzy. Może zostać zdefiniowany automatycznie przez następujący wzór: $\alpha^t = -3 * \frac{t}{N} + 3.5$
 - d. beta – wartość z zakresu R^+ , parametr definiujący poziom zaufania cząstki do wiedzy zapewnianej przez swoich sąsiadów. Może zostać zdefiniowany automatycznie przez następujący wzór: $\beta^t = 3 * \frac{t}{N} + 0.5$
 - e. w – wartość z zakresu R^+ , parametr definiujący zdolność roju do zmiany kierunku ruchu (zwany też masą bezwładności) czyli do opuszczania znalezionych przez cząstki ekstremów lokalnych. Może zostać zdefiniowany automatycznie przez następujący wzór: $w^t = 0.4 * \frac{(t-N)}{N^2} + 0.4$
3. Zdefiniuj populację cząstek, gdzie każda cząstka posiada poniższą właściwość:
 - a. P - sprawność cząstki w chwili t ustalana losowo na początku symulacji
 - b. V – prędkość cząstki w chwili t ustalana losowo na początku symulacji
 - c. r_1 – wartość z zakresu $[0, 2]$ definiująca poziom zaufania cząstki we własne zdolności poznawcze ustalana losowo na początku symulacji
 - d. r_2 – wartość z zakresu $[0, 2]$ definiująca poziom zaufania cząstki do wiedzy społeczności oraz ustalana losowo na początku symulacji
4. Ustal warunek/ki zatrzymania
 - a. osiągnięcie maksymalnej liczby iteracji
 - b. wykrycie zatrzymanie się cząstek
5. Powtarzaj poniższe kroki dla każdej cząstki w każdym kroku iteracji
 - a. Oblicz sprawność cząstki o indeksie i: $P_i^{t+1} = P_i^t + V_i^{t+1}$
 - b. Oblicz prędkość cząstki o indeksie i zgodnie z poniższym wzorem:

$$V_i^{t+1} = w * V_i^t + \alpha * r_1(P_{best(i)}^t - P_i^t) + \beta * r_2(P_{bestglobal}^t - P_i^t)$$
 - c. Po zaktualizowaniu wartości całej populacji sprawdź czy warunek zatrzymania został osiągnięty

3.6.2 Modyfikacja PSO dla problemu TSP

Algorytm roju cząstek świetnie nadaje się do poszukiwania ekstremum funkcji wielu zmiennych, gdzie alternatywne rozwiązanie stanowi wektor $[x_1, x_2, x_3 \dots x_n]$ a wektor prędkości $[v_1, v_2, v_3 \dots v_n]$ a suma tych dwóch wektorów stanowi nowe położenie naszej cząstki w przestrzeni dostępnych rozwiązań. Niestety by zastosować PSO do problemu kombinatorycznego takiego jak TSP pierwotny algorytm należy zmodyfikować.

W celu dostosowania PSO do problemu TSP autor pracy postanowił zrezygnować ze współczynników w , r_1 , r_2 oraz za P przyjąć rozwiązanie problemu TSP jako jedną z permutacji N miast. Głównym problemem dostosowania okazuje się jednak sposób zdefiniowania prędkości każdej cząstki oraz sposobu jej modyfikacji zależnej od rozwiązania najlepszego i parametrów algorytmu tak by poniższe bliżej nie określone jeszcze działania pozwoliły zapewnić nowe akceptowalne rozwiązanie problemu TSP zbliżone do najlepszego znalezione w chwili t przez inne cząstki.

$$P_i^t \odot V_i^t = P_i^{t+1}$$

$$F(P_i^t, P_{best(i)}^t, P_{globalbest}^t, \alpha, \beta, V_i^t) = V_i^{t+1}$$

Autor pracy w celu rozwiązania powyższego problemu postanowił wykorzystać operator krzyżowania typu Swap z prawdopodobieństwem równym wartości parametru α oraz β . Parametr α nadal odpowiada za ufność cząstki we własne funkcje poznawcze a parametr β nadal za ufność cząstki w najlepsze rozwiązanie znalezione przez populację, ale ich zakres od teraz będzie wynosił $[0, 1]$. Wektor prędkości stanowił będzie listę o maksymalnej długości $2N$ (mniejsza wartość, gdy rozwiązanie cząstki pokrywa się w pewnym stopniu z rozwiązaniem wypracowanym przez rój) która będzie zawierała obiekty przechowujące instrukcję jak wykonać operację zamiany elementów w rozwiązaniu prezentowanym przez cząstkę rozwiązaniu. Autor postanowił, że wektor prędkości będzie liczył maksymalnie $2N$ elementów, ponieważ pierwsze około N elementów będzie zawierało instrukcję jak z prawdopodobieństwem równym α przeprowadzić operację zamiany 1 elementowej by zbliżyć się nią do najlepszego rozwiązania osiągniętego przez obecnie rozpatrywaną cząstkę. Natomiast reszta wektora V będzie zawierać instrukcję jak z prawdopodobieństwem β przeprowadzić funkcję swap tak by rozwiązanie reprezentowane przez daną cząstkę ulepszyć na wzór najlepszego rozwiązania wypracowanego przez cały

rój. Ostatnim elementem jest zdefiniowanie operacji, która pozwoli wypracować nowe rozwiązanie na bazie starego oraz informacji zawartych w wektorze prędkości. Autor uznał, że tą operacją będzie poddanie rozwiązania obecnego wypracowanego przez cząstkę działaniu serii operacji zamiany pojedynczego elementu wtedy i tylko wtedy, gdy wylosowane prawdopodobieństwo dla operacji zamiany będzie mniejsze niż to które zostało zawarte w instrukcji do funkcji zamiany w wektorze V. Po zaktualizowaniu wartości rozwiązania dla cząstki następuje wyczyszczenie wektora prędkości dla niej by w kolejnej iteracji mogła nastąpić jego ponowna redefinicja. Poniżej znajduje się fragment kodu reprezentujący powyżej opisane modyfikacje.

```

55     def run(self):
56
57         # for each time step (iteration)
58         for t in range(self.iterations):
59
60             # updates gbest (best particle of the population)
61             self.gbest = min(self.particles, key=attrgetter('cost_pbest_solution'))
62
63             # for each particle in the swarm
64             for particle in self.particles:
65
66                 particle.clearVelocity() # cleans the speed of the particle
67                 temp_velocity = []
68                 solution_gbest = copy.copy(self.gbest.getPBest()) # gets solution of the gbest
69                 solution_pbest = particle.getPBest()[:] # copy of the pbest solution
70                 solution_particle = particle.getCurrentSolution()[
71                     : ] # gets copy of the current solution of the particle
72
73                 # generates all swap operators to calculate (pbest - x(t-1))
74                 for i in range(self.graph.amount_vertices):
75                     if solution_particle[i] != solution_pbest[i]:
76                         # generates swap operator
77                         swap_operator = (i, solution_pbest.index(solution_particle[i]), self.alfa)
78
79                         # append swap operator in the list of velocity
80                         temp_velocity.append(swap_operator)
81
82                         # makes the swap
83                         aux = solution_pbest[swap_operator[0]]
84                         solution_pbest[swap_operator[0]] = solution_pbest[swap_operator[1]]
85                         solution_pbest[swap_operator[1]] = aux
86
87                 # generates all swap operators to calculate (gbest - x(t-1))

```

Rysunek 5 Fragment implementacji PSO dla TSP reprezentujący tworzenie 1 części wektora prędkości

```

87         # generates all swap operators to calculate (gbest - x(t-1))
88         for i in range(self.graph.amount_vertices):
89             if solution_particle[i] != solution_gbest[i]:
90                 # generates swap operator
91                 swap_operator = (i, solution_gbest.index(solution_particle[i]), self.beta)
92
93                 # append swap operator in the list of velocity
94                 temp_velocity.append(swap_operator)
95
96                 # makes the swap
97                 aux = solution_gbest[swap_operator[0]]
98                 solution_gbest[swap_operator[0]] = solution_gbest[swap_operator[1]]
99                 solution_gbest[swap_operator[1]] = aux
100
101             # updates velocity
102             particle.setVelocity(temp_velocity)
103
104             # generates new solution for particle
105             for swap_operator in temp_velocity:
106                 if random.random() <= swap_operator[2]:
107                     # makes the swap
108                     aux = solution_particle[swap_operator[0]]
109                     solution_particle[swap_operator[0]] = solution_particle[swap_operator[1]]
110                     solution_particle[swap_operator[1]] = aux
111
112             # updates the current solution
113             particle.setCurrentSolution(solution_particle)
114             # gets cost of the current solution
115             cost_current_solution = self.graph.getCostPath(solution_particle)
116             # updates the cost of the current solution
117             particle.setCostCurrentSolution(cost_current_solution)
118
119             # checks if current solution is pbest solution
120             if cost_current_solution < particle.getCostPBest():
121                 particle.setPBest(solution_particle)
122                 particle.setCostPBest(cost_current_solution)

```

Rysunek 6 Fragment implementacji PSO dla TSP definiujący 2 część wektora prędkości oraz tworzący nowe lepsze rozwiązanie na bazie wiedzy cząstki i roju

4. Plan pracy

Praca badawcza zawsze wymaga od badacza starannego przygotowania. Nie inaczej jest również w przypadku tej pracy. W celu przeanalizowania działania algorytmów heurystycznych dla symetrycznego problemu TSP naukowiec musiał najpierw nie tylko zapoznać się z istniejącymi już pracami omawiających podobne zagadnienia, ale również musiał biorąc pod uwagę ograniczenia sprzętowe rozważyć jakie i jak przygotować dane wejściowe dla zaimplementowanych algorytmów a także przygotować zestaw pytań, na które wyniki prowadzonych badań powinny dać odpowiedzi.

4.1 Przygotowanie danych do badań

Biorąc pod uwagę, że do przeprowadzenia pomiarów potrzebny jest dość spory zbiór danych wejściowych, by móc jak najbardziej zminimalizować wpływ błędów pomiarowych na otrzymane wyniki. Autor pracy postanowił wygenerować od 4-15 losowo rozmieszczonych punktów (miast) w układzie kartezjańskim o rozmiarach 2000 x 2000 jednostek, upakowanych w paczki plików po 100. Każdy tak wygenerowany plik nazywany w dalszej części próbką musiał sprostać szeregu wymaganiom, które naukowiec postanowił poniżej opisać:

Wymagania próbki pomiarowej:

- każdy plik wejściowy zawiera dane wejściowe w formacie JSON
- każdy plik zawiera miasto znajdujące się w początku układu współrzędnych (w punkcie $[x:0,0; y:0,0]$)
- każdy z plików będzie dodatkowo zawierał następujące informacje statystyczne na temat położenia miast:
 - maksymalna wartość x oraz y
 - minimalna wartość x oraz y
 - wartość średnia dla x oraz y
 - odchylenie standardowe dla x oraz y
 - wartość pierwszego kwartylu ($Q1$) dla x oraz y
 - wartość drugiego kwartylu ($Q2$, mediana) dla x oraz y
 - wartość trzeciego kwartylu ($Q3$) dla x oraz y
- komiwojażer zawsze będzie wyruszał z początku układu współrzędnych tj. z miasta o lokalizacji w punkcie $(x:0,0; y:0,0)$

Dane statystyczne dołączone do każdej próbki mają na celu pomóc wykryć korelację między rozrzutem wygenerowanych miast a dużym odstępstwem pomiaru zrealizowanego dla danej próbki w odniesieniu do średnich wyników całej grupy próbek zawężonej do danej liczby miast oraz użytego do rozwiązania problemu algorytmu i danych wartości parametrów z jakimi został przeprowadzony pomiar.

4.2 Metryki pomiarowe

Jedną z najważniejszych aspektów pracy badawczej jest zdefiniowanie metryk jakich naukowiec zamierza użyć w ramach, których zamierza wykonywać pomiary. Problem TSP należy do problemów NP-trudnych więc naturalną rzeczą jest by określić dobroć/optymalność rozwiązań uzyskanych przez algorytmy heurystyczne dla każdej próbki naukowiec musiał poznać rozwiązanie optymalne problemu (sekwencja odwiedzanych miast spełniająca wymagania i charakteryzująca się najniższym kosztem). Do wyznaczenia optymalnego rozwiązania został przez autora wykorzystany omówiony wcześniej algorytm Helda-Karpa zrealizowany zgodnie z ideą programowania dynamicznego.

Wszystkie algorytmy również heurystyczne, pomimo że prezentują różne strategie rozwiązywania danego problemu zawsze do ich uruchomienia wykorzystywany jest komputer, którego najdroższymi zasobami są CPU używane do przeprowadzania obliczeń oraz pamięć operacyjna używana do ich przechowywania, a dla nas ludzi najistotniejszy jest czas wykonania się programu.

Podsumowując, wszystkie powyżej omówione metryki zostały zaprezentowane w poniższym zestawieniu:

Metryki pomiarowe wykorzystane podczas badań:

- czas wykonania algorytmu mierzony w sekundach
- minimalna wartość zużycia procesora w procentach
- średnia wartość zużycia procesora w procentach
- maksymalna wartość zużycia procesora w procentach
- ilość zaalokowanej przez algorytm pamięci RAM mierzona w bajtach
- błąd bezwzględny otrzymanego rozwiązania (koszt rozwiązania alternatywnego – koszt rozwiązania optymalnego)
- błąd względny otrzymanego rozwiązania ($[\text{błąd bezwzględny} / \text{koszt rozwiązania optymalnego}] * 100\%$)

5. Cele badań

Każdy naukowiec zanim przystąpi do zbierania danych pomiarowych powinien zawsze odpowiedzieć sobie na pytanie co i dlaczego mierzy, na jakie pytania wyniki badań mają przynieść odpowiedzi. Autor tej pracy również je zadał i zostały one przedstawione poniżej w ramach rozbudowanej listy punktów.

1. Która z dwóch dedykowanych heurystyk dla algorytmu A* pozwala osiągnąć lepsze wyniki mniejszym nakładem czasu, pamięci i CPU
 - heurystyka A – iloczyn krawędzi pozostałych do wykorzystania $(n - k)$ i wagi najtańszej krawędzi
 - heurystyka B – suma $(n - k)$ najmniejszych wag niewykorzystanych jeszcze krawędzi
2. Która z badanych bibliotek oferująca rozwiązanie problemu TSP z użyciem algorytmu genetycznego jest lepsza w użyciu i z jakim zestawem parametrów
 - zakres bibliotek
 - i. biblioteka scikit-opt
 - ii. biblioteka mlrose
 - optymalny dobór parametrów algorytmu genetycznego
 - i. rozmiar populacji: 100
 - ii. prawdopodobieństwo mutacji [0.001, 0.01, 0.1] (wspierany przez obie implementacje)
 - iii. ilość podejść by móc znaleźć lepsze rozwiązanie w każdej epoce [1, 5, 10] (wspierany przez obie implementacje)
 - iv. maksymalna liczba iteracji [100, 200, 300] (wspierany przez obie implementacje)
 - v. operator krzyżowania: krzyżowanie typu PMX (tylko scikit-opt)
 - vi. operatory mutacji:
 1. mutacja typu TSP_1
 2. mutacja odwracająca sekwencję (domyślna)
 - vii. typy selekcji (tylko scikit-opt)
 1. selekcja turniejowa szybka (domyślna)
 2. selekcja kołem ruletki typu 1

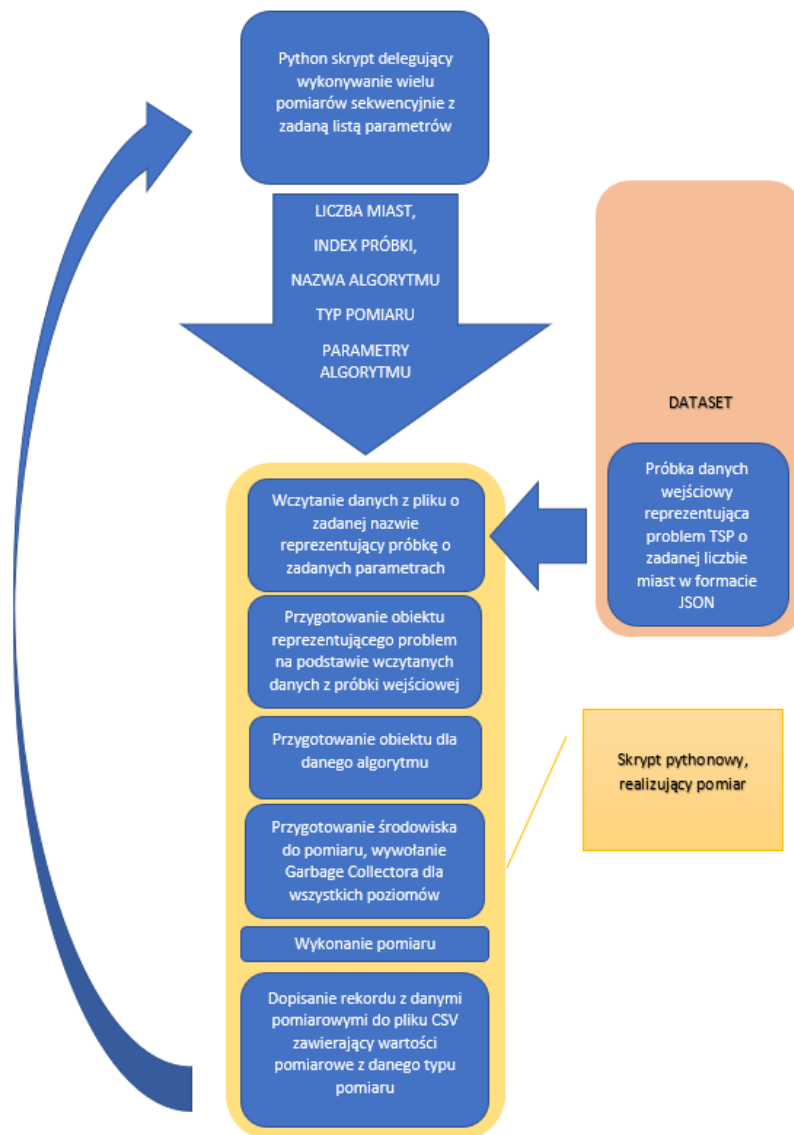
3. selekcja kołem ruletki typu 2

viii. ilość rund (tylko scikit-opt) zakres [1, 2, 3]

3. Potwierdzić czy algorytm NN (greedy search) zawsze niskim kosztem zapewni nam rozwiązanie nie gorsze niż 20% od optymalnego rozwiązania
4. Która wartość współczynnika alfa z $[0.1 - 0.9]$ co 0.1 zapewni nam najlepsze rozwiązanie z użyciem implementacji algorytmu symulowanego wyżarzania z biblioteki python-tsp przy wykorzystaniu algorytmu „2-opt” do zmiany struktury rozwiązania
5. Określić optymalny zestaw parametrów dla zaimplementowanego przez autora algorytmu PSO, który pozwoli uzyskać jak najlepsze rozwiązania zmieniając dane parametry w ramach poniższych parametrów
 - maksymalna liczba iteracji: 100
 - rozmiar populacji: 100
 - wartość parametru alfa $[0.1 - 0.9]$
 - wartość parametru beta $[0.1 - 0.9]$
6. Określić optymalny zestaw parametru dla zaimplementowanego przez autora algorytmu ACO, który pozwoli uzyskać jak najlepsze rozwiązania zmieniając dane parametry w ramach poniższych parametrów
 - maksymalna liczba iteracji 20
 - rozmiar populacji 100
 - wartość parametru alfa $0.5 - 3.0$ co 0.5
 - wartość parametru beta $0.5 - 3.0$ co 0.5
 - wartość parametru rho $[0.1 - 0.9]$
7. Podsumować który algorytm heurystyczny pozwala nam dość dobrze rozwiązać symetryczny problem TSP, jak najniższym kosztem zużycia CPU i pamięci operacyjnej dla wybranego najlepszego dopasowania parametrów do algorytmów przez autora

6. Architektura pomiarowa

Do przeprowadzenia badań i zebrania pomiarów autor wykorzystał język wysokiego poziomu python, który wspiera paradygmat programowania obiektowego. Pozwoliło to autorowi opracować szereg przydatnych klas i skryptów realizujących określone zadania dzięki czemu operacje odczytywania, parsowania plików wejściowych JSON oraz tworzenie plików CSV zawierających dane pomiarowe a także realizacja samych pomiarów z użyciem różnych algorytmów mogła zostać zaimplementowana z wykorzystaniem wzorców projektowych. Poniżej został przedstawiony schemat architektury pomiarowej zrealizowany w oparciu o dwa główne skrypty pythonowe.



Rysunek 7 Schemat architektury pomiarowej

Powyższy rysunek przedstawia sposób w jaki autor pracy opracował architekturę pomiarową dzieląc ją na dwa skrypty pythonowe (po uruchomieniu 2 procesy delegujący i pomiarowy) gdzie jeden wywołuje drugi z określonymi parametrami. Jednakże sam sposób realizacji pomiarów CPU pamięci został przedstawiony na poniższym diagramie BPMN.

Na powyższym diagramie, można zauważyć że pomiar CPU został zrealizowany przez powołanie przez wątek pomiarowy przed wykonaniem pomiaru wątku CPU profilera odpowiedzialnego za odczytywanie stanu zużycia procesora w 1 sekundowych odstępach czasu przy wykorzystaniu biblioteki psutil [12], i zbierania ich w instancji klasy DataCollector dopóki wątek pomiarowy nie stwierdzi że pomiar może zostać zakończony i wypełniony danymi kolektor nie zostanie przekazany do wątku pomiarowego a wątek CPU profilera ulegnie zabiciu. Poniżej zamieszczony został kod klasy CpuProfiler.

```
12 class CpuProfiler(threading.Thread):
13     def __init__(self):
14         super().__init__()
15         self.interval_in_seconds = 1
16         self.collector = DataCollector()
17         self.stopped = False
18         self.after = time.perf_counter()
19         self.collector.add_data(UTILIZATION_OF_CPU, list())
20         self.start_time = None
21
22     def run(self):
23         initialized = False
24         self.start_time = time.clock()
25         while not self.stopped:
26             if not initialized:
27                 self.make_measure()
28                 initialized = True
29                 self.after = time.perf_counter() + self.interval_in_seconds
30             else:
31                 if time.perf_counter() > self.after:
32                     self.make_measure()
33                     self.after = time.perf_counter() + self.interval_in_seconds
34             if self.stopped:
35                 break
36
37     def make_measure(self):
38         if not self.stopped:
39             self.collector.add_data_to_list(UTILIZATION_OF_CPU, psutil.cpu_percent(0.1))
40
41     def stop(self):
42         self.stopped = True
43         stop = time.clock()
44         self.collector.add_data(TIME_DURATION_IN_SEC,
45                               stop - self.start_time)
46
47     def get_collector(self):
48         fields = [UTILIZATION_OF_CPU]
49         for field in fields:
50             if len(self.collector.dictionary_of_data[field]) == 0:
51                 self.collector.dictionary_of_data[field] = DEFAULT_VALUE
52         series = pd.Series(self.collector.dictionary_of_data[UTILIZATION_OF_CPU])
53         self.collector.dictionary_of_data[MIN_UTILIZATION_OF_CPU] = series.min()
54         self.collector.dictionary_of_data[AVG_UTILIZATION_OF_CPU] = series.mean()
55         self.collector.dictionary_of_data[MAX_UTILIZATION_OF_CPU] = series.max()
56         self.collector.dictionary_of_data[STD_UTILIZATION_OF_CPU] = series.std()
57         return self.collector
```

Rysunek 8 Kod źródłowy klasy CpuProfiler

Dzięki temu, że Python jest dynamicznie typowany oraz nie pozwala programistom bezpośrednio na zarządzanie procesem alokowania i zwalniania pamięci operacyjnej, realizacja pomiaru zaalokowanej pamięci przez dany segment kodu nie należy do operacji trywialnych. Dodatkowym problemem dokładnego pomiaru zaalokowanej i zwalnianej pamięci jest fakt, że Python podobnie jak i Java posiada zaimplementowany mechanizm odśmiechania pamięci, który w wyniku spontanicznego uruchomienia może zapewnić, że podczas pomiaru zużycia pamięci w podobny sposób jak to miało miejsce w przypadku CPU będziemy zauważyć, że zebrane pomiary nie posiadają wyraźnego trendu wzrostowego a raczej podlegają stopniowym fluktuacjom. W celu wykluczenia tego zjawiska, do pomiarów zużycia pamięci przez dany algorytm w bajtach autor wykorzystał bibliotekę `tracemalloc` [11], która pozwala na dokładniejszy pomiar zużycia pamięci, ponieważ jest to biblioteka specjalnie zaprojektowana do śledzenia bloków pamięci przydzielonych przez Pythona.

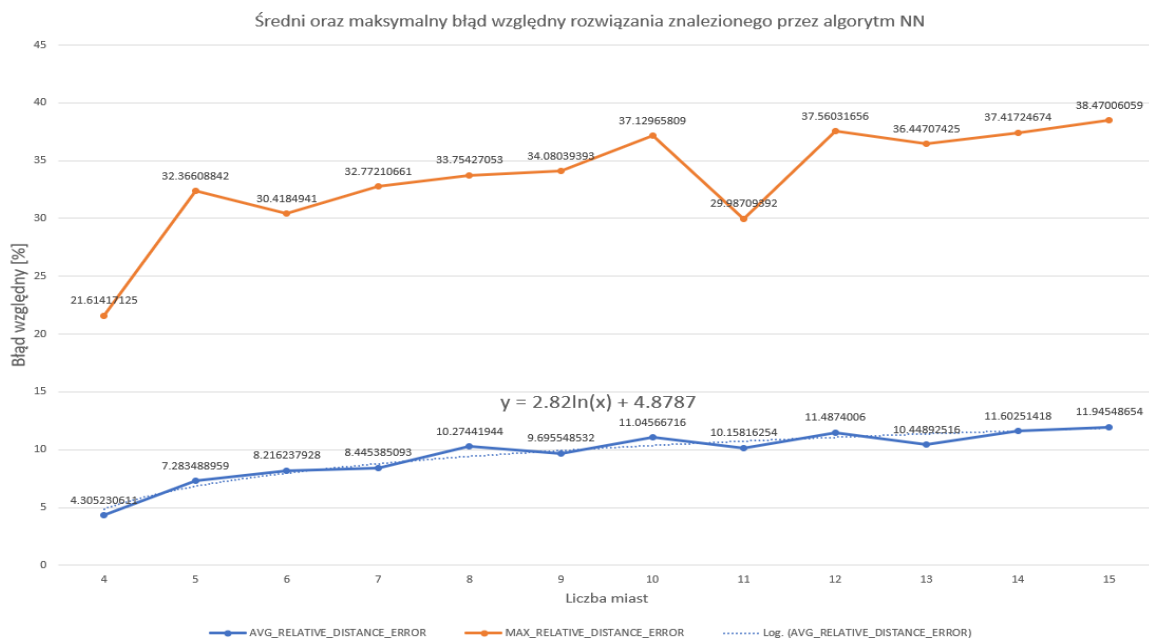
```
66     def start_counting_with_time_and_trace_malloc(self) -> DataCollector:
67         self.can_be_run()
68         collector = DataCollector()
69         self.clear_data_before_measurement()
70
71         tracemalloc.start()
72
73         before_size, before_peak = tracemalloc.get_traced_memory()
74         start = time.clock()
75
76         _, _ = self.find_way()
77
78         stop = time.clock()
79         after_size, after_peak = tracemalloc.get_traced_memory()
80         tracemalloc.stop()
81
82         collector.add_data(TIME_DURATION_IN_SEC, stop - start)
83         collector.add_data(USED_MEMORY_BEFORE_MEASUREMENT_IN_BYTES, before_size)
84         collector.add_data(USED_MEMORY_PEAK_BEFORE_MEASUREMENT_IN_BYTES, before_peak)
85         collector.add_data(USED_MEMORY_AFTER_MEASUREMENT_IN_BYTES, after_size)
86         collector.add_data(USED_MEMORY_PEAK_AFTER_MEASUREMENT_IN_BYTES, after_peak)
87         collector.add_data(USED_MEMORY_DIFF_BEFORE_AFTER_MEASUREMENT_IN_BYTES,
88                             after_size - before_size)
89         collector.add_data(USED_MEMORY_PEAK_DIFF_BEFORE_AFTER_MEASUREMENT_IN_BYTES,
90                             after_peak - before_peak)
91         collector.add_data(PARAMETERS, self.config)
92         return collector
93
```

Rysunek 9 Fragment kodu źródłowego odpowiedzialnego za pomiar zużycia pamięci z użyciem biblioteki `trace malloc`

7. Przegląd wyników badań

7.1 Analiza algorytmu najbliższego sąsiada

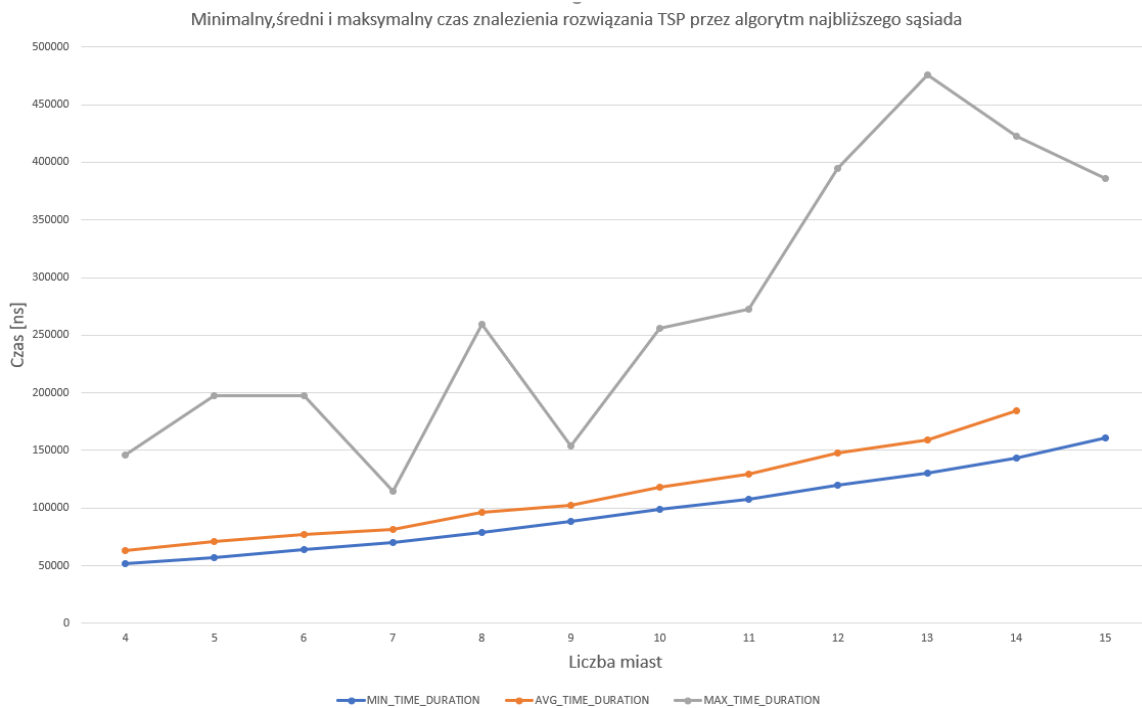
Algorytm NN (greedy search) jako jeden z najprostszych i najszybszych metaheurystyk nie pozwala osiągnąć zbyt dobrych rozwiązań. Zgodnie z danymi z Wikipedii jakość otrzymywanych z jego udziałem rozwiązań jest średnio gorsza o 25% od optymalnych rozwiązań. Autor pracy zamierza to stwierdzenie poddać pod weryfikację, przeprowadzając własne badania dla 1200 próbek po 100 dla liczby miast od 4 do 15.



Rysunek 10 Wykres obrazujący średni i minimalny błąd uzyskanych rozwiązań z użyciem algorytmu NN

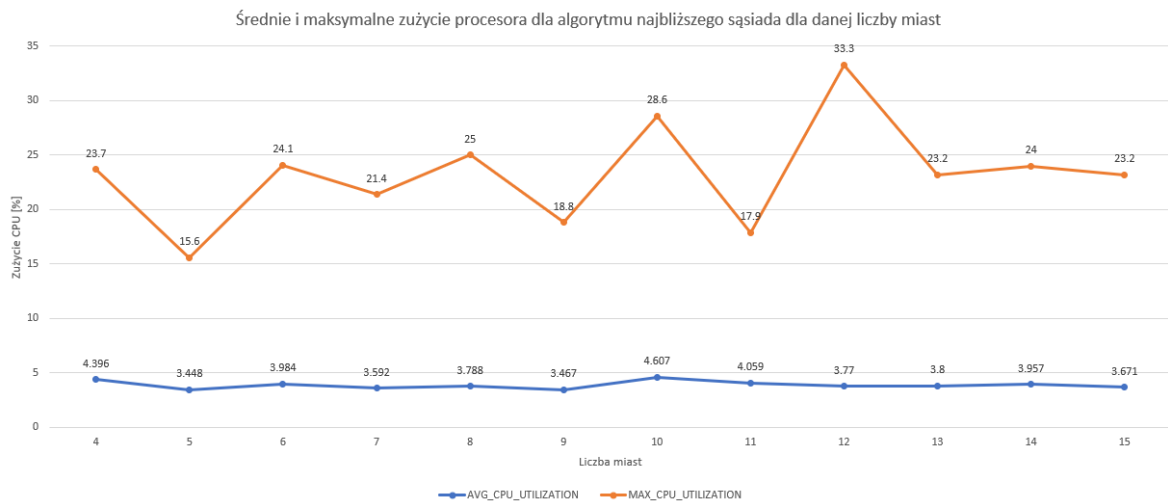
Zgodnie z danymi prezentowanymi na powyższym diagramie rozwiązania problemu TSP z wykorzystaniem algorytmu TSP, rzeczywiście są średnio odległe od rozwiązania optymalnego o 25%. Podczas badań wyszło, że funkcją, która najlepiej aproksymuje średni błąd względny dla zadanej liczby miast jest funkcja logarytmiczna. Pomimo, że średni błąd względny otrzymanych rozwiązań mieści się w granicach od 5 do 12,5% to jednak po sprawdzeniu maksymalnych wartości błędu względnego zależnego od ilości miast sytuacja wygląda już mniej optymistycznie.

Według badań autora maksymalna wartość błędu względnego od 4 do 15 miast oscyluje w granicach od 21 do 38,5% i prawdopodobnie utrzymuje nadal tendencję wzrostową dla większej liczby miast.



Rysunek 11 Wykres minimalnego, średniego i maksymalnego czasu znalezienia rozwiązania TSP przez algorytm NN

W wyniku zbadania czasu wykonania algorytmu, została wykryta duża nieregularność krzywej, która reprezentuje maksymalny czas wykonania w sekundach dla zadanej liczby miast. Powyższą niemonotoniczność na wykresie można wytłumaczyć tym, że w przypadku poszukiwania miast z minimalną odległością użyto dwóch przerywanych pętli for co sprawia, że nie za każdym razem zostanie wykonanych n^2 iteracji. To, że maksymalny czas znalezienia rozwiązania dla danej liczby miast jest bliższy wartości średniej niż inne punkty wykresu jest zasługą wykonania mniejszej liczby iteracji przez pominięcie liczenia dystansu do już odwiedzonych miast. Poniższe wykresy obrazujące zużycie CPU zdają się potwierdzać powyższe wyjaśnienie.



Rysunek 12 Wykres średniego i maksymalnego zużycia CPU przez algorytm NN

Dane zobrazowane na powyższym wykresie wskazują, że przez to, iż algorytm najbliższego sąsiada jest bardzo naiwną a zarazem prostą do zaimplementowania heurystyką to średnie zużycie procesora niezależnie od wielkości przestrzeni możliwych rozwiązań nie pozostaje większe niż 5% zużycia CPU. Pomimo że średnie zużycie CPU jest niskie to podczas działania algorytmu występują momenty, gdy zużycie CPU może wzrosnąć nawet do 33,3%.

7.2 Analiza algorytmu A*

Algorytm A* pozwala na dokonywanie przeszukiwania przestrzeni rozwiązań symetrycznego problemu komiwojażera w sposób bardziej inteligentny niż algorytm NN. Dzięki zastosowaniu połączenia kosztu oraz wartości heurystyki rozwiązanie to umożliwia modyfikację ostatnio znalezionej sekwencji miast tak by z każdą kolejną iteracją przybliżać się do rozwiązania optymalnego. Heurystyk stanowi w nim sposób wyliczania wskazówki jak odróżnić rozwiązania lepsze od obecnych o tych gorszych i diametralnie zmienić drogę komiwojażera. Ze względu na długi czas obliczeń potrzebny do przeprowadzenia poniższych badań autor pracy zdecydował zmniejszyć liczbę badanych przypadków testowych z 4-15 na 4-13 miast co sprawiło, że ilość zbadanych próbek spadła z 1200 do 1000 odrębnych przypadków testowych po 100 dla każdej liczby miast z zakresu 4-13.

W ramach przeprowadzonych badań, autor chciał na podstawie ich wyników odpowiedzieć na pytanie która z heurystyk A czy B jest lepsza porównując kolejno zużycie

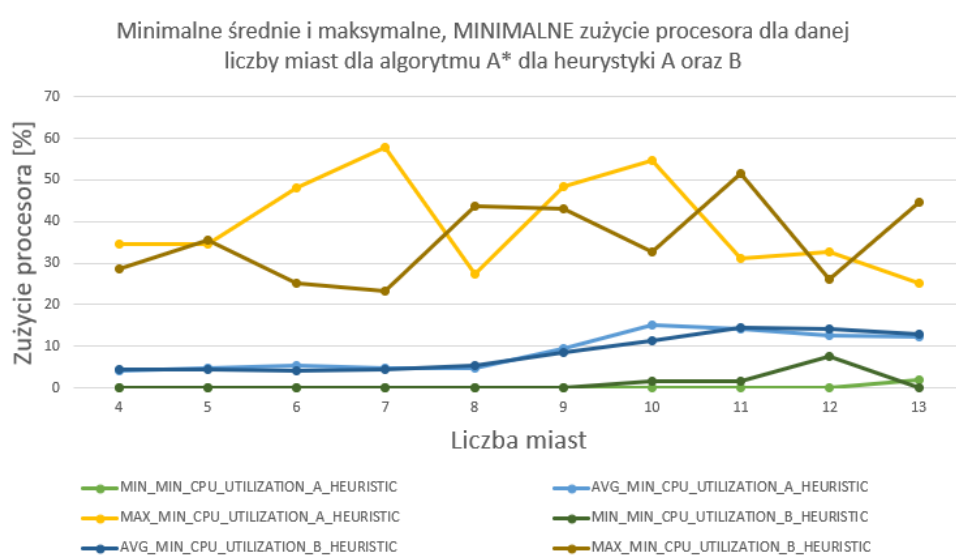
procesora, pamięci RAM. czas wykonanie obliczeń oraz minimalny, średni oraz maksymalny błąd względny uzyskanych rozwiązań dla danej liczby miast.

Badane heurystyki:

1. A – iloczyn krawędzi pozostałych do wykorzystania ($n - k$) i wagi najtańszej krawędzi
2. B – suma ($n - k$) najmniejszych wag niewykorzystanych jeszcze krawędzi

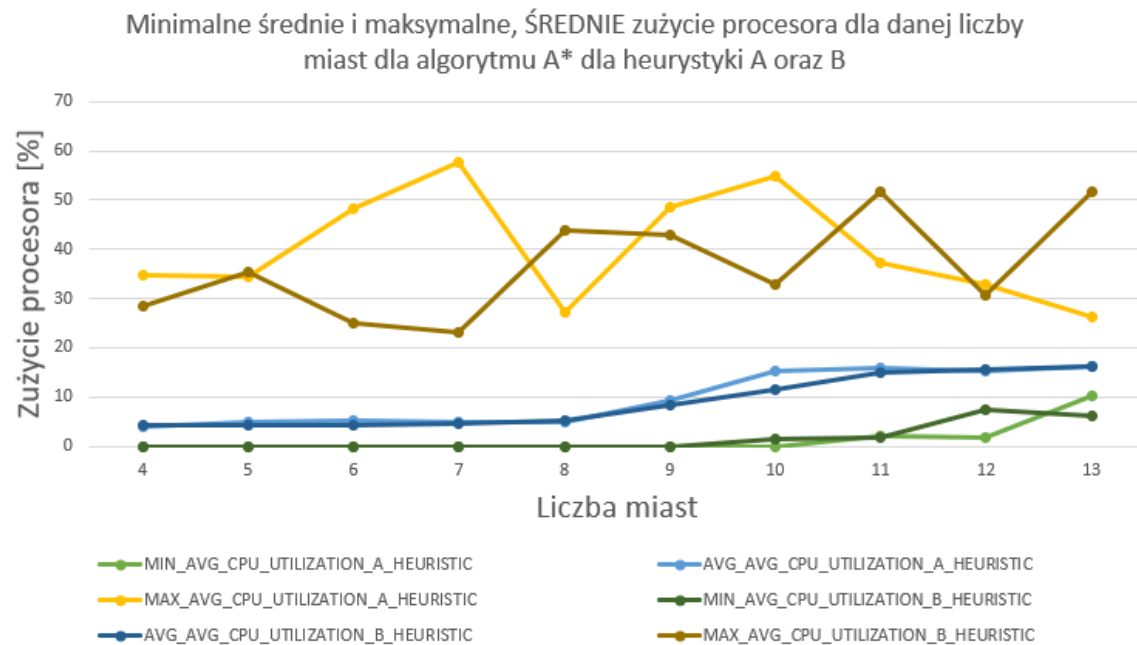
7.2.1 A* wyniki pomiarów zużycia CPU

Analiza zużycia CPU pozwoli odpowiedzieć na pytanie która heurystyka potrzebuje więcej czasu oraz zasobów obliczeniowych w celu ustalenia z użyciem algorytmu A*. Przeanalizowanie zarejestrowanego minimalnego średniego i maksymalnego zużycie CPU dla każdej ilości miast (100 próbek) gdzie dla każdej próbki zostało zarejestrowane minimalne, średnie oraz maksymalne zużycie CPU pozwoli określić czy wzrost liczby miast i maksymalne zużycie CPU wykazują korelację. Aby lepiej zrozumieć dane zobrazowane na poniższych wykresach należy wspomnieć, że dla każdego pomiaru dla danej liczby miast i próbki o indeksie od 0-100 został wykonany pomiar zużycia CPU, który przebiegał w oparciu o kolekcjonowanie zużycia CPU w trakcie rozwiązywania problemu w różnych odstępach czasu wynoszących 1s. Dla każdej w ten sposób utworzonej kolekcji została wyliczona wartość minimalna średnia oraz maksymalna dla danej próbki po czym posiadając tego rodzaju dane zebrane dla każdej próbki dla danej liczby miast w problemie została

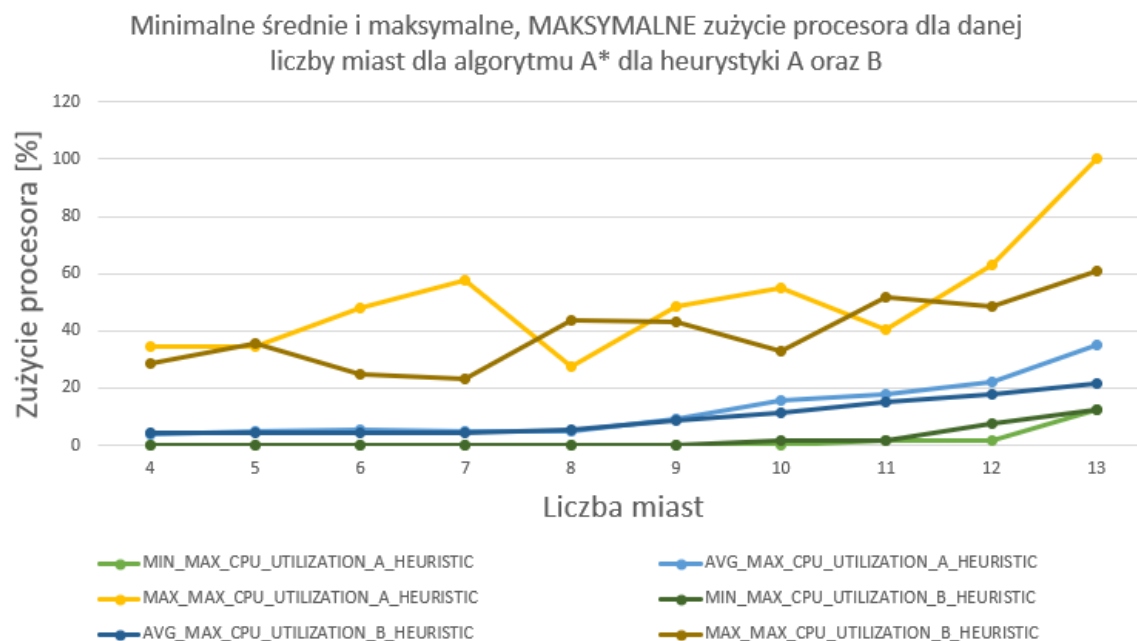


Rysunek 13 Wykres minimalnej, średniej oraz maksymalnej wartości zarejestrowanego minimalnego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk

znaleziona wartość minimalna, maksymalna oraz wyliczona średnia wartość i to właśnie one są prezentowane na poniższych wykresach.



Rysunek 14 Wykres minimalnej, średniej i maksymalnej wartości zarejestrowanego średniego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk



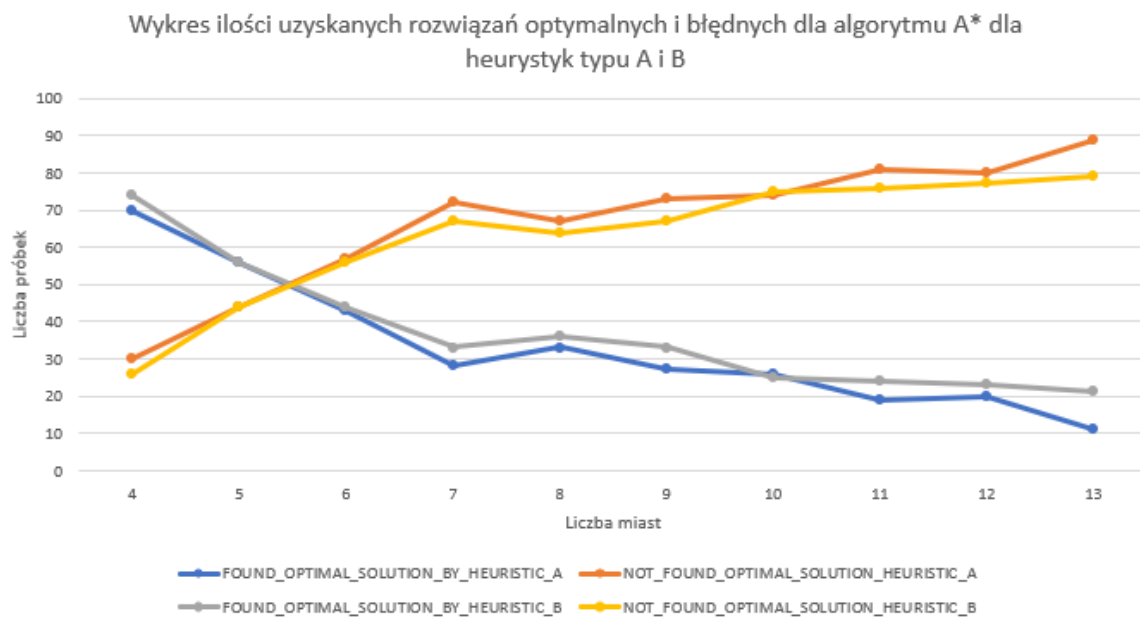
Rysunek 15 Wykres minimalnej, średniej i maksymalnej wartości zarejestrowanego maksymalnego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk

Dane przedstawione na powyższych wykresach jednoznacznie stwierdzają, że zapotrzebowanie algorytmu A* z heurystyką A jest większe niż użycie tego samego algorytmu z heurystyką B. Wykresy przedstawiające minimum i średnią dla maksymalnego % zużycia procesora dla obu metod dla danej liczby miast podlegają fluktuacjom co jednoznacznie wskazuje, że sama ilość miast nie jest jedyną zmienną, od której zależy estymowane zapotrzebowanie algorytmu na potrzebną moc obliczeniową CPU.

Maksymalnego zapotrzebowania algorytmu A* z użyciem heurystyki typu A, zdaje się charakteryzować większym zapotrzebowaniem CPU wraz ze wzrostem ilości miast niż algorytm A* z wykorzystaniem heurystyki typu B. By stwierdzić czy heurystyka typu B jest lepsza od A należy również porównać wartość uzyskanych błędów oraz zapotrzebowanie obu metod na pamięć operacyjną.

7.2.2 A* analiza błędów uzyskanych rozwiązań

Algorytm A* w zasadzie jest jak algorytm Dijkstry (zapewnia znalezienie optymalnego rozwiązania problemu, ponieważ wartość heurystyki = 0, zostaje sama funkcja kosztu), tylko że w jego przypadku rozwiązanie zależy nie tylko od realnego kosztu rozwiązania, ale również od wzoru wykorzystanej heurystyki. Poniżej przedstawione zostały wyniki porównania dokładności oraz ilości znalezionych rozwiązań z wykorzystaniem heurystyki typu A oraz B.

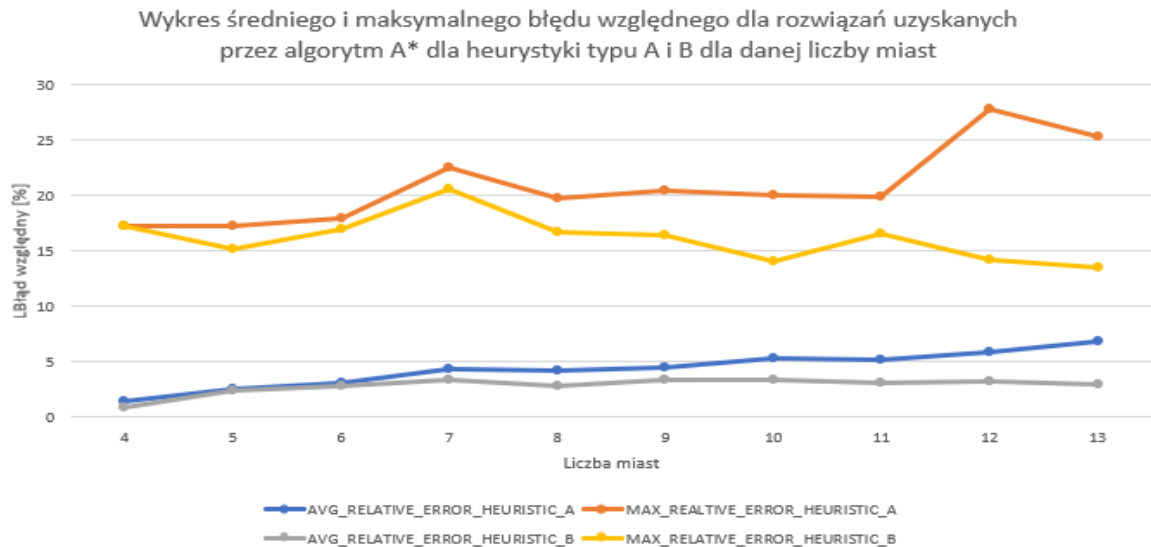


Rysunek 16 Wykres ilości uzyskanych rozwiązań optymalnych do błędnych dla danej liczby miast i obu badanych heurystyk

Dane przedstawione na powyższym wykresie, pozwalają zaobserwować dwa fakty:

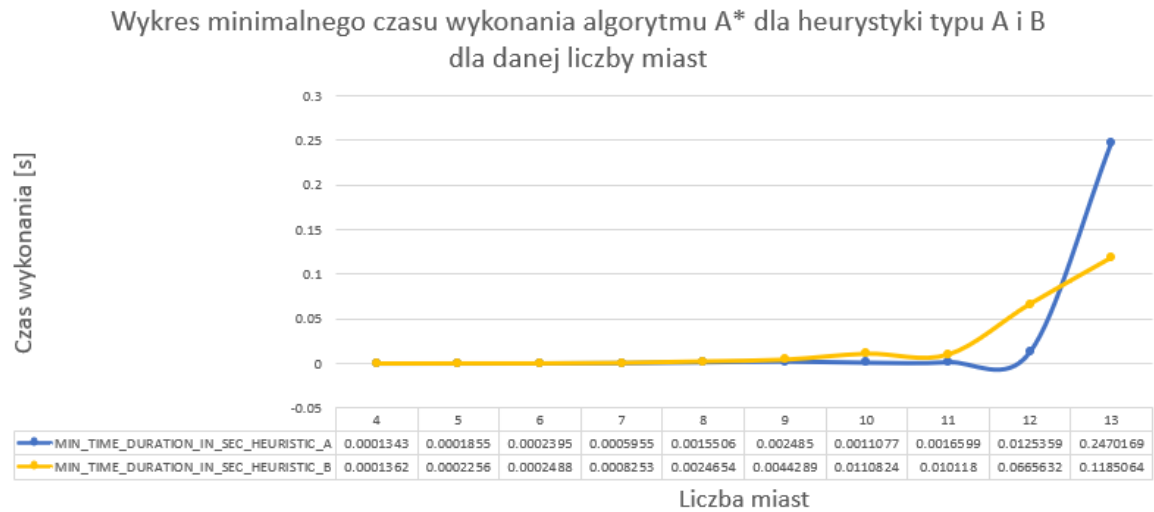
- najszybszy spadek efektywności algorytm A* został zarejestrowany dla problemów liczących od 4 do 7 miast
- heurystyka typu B pozwoliła znaleźć więcej optymalnych rozwiązań dla tych samych problemów niż heurystyka typu A

W celu uzyskania dokładniejszych danych zostało zbadane średnie oraz maksymalny błąd uzyskanych rozwiązań znalezionych przez algorytm A* dla obu heurystyk.

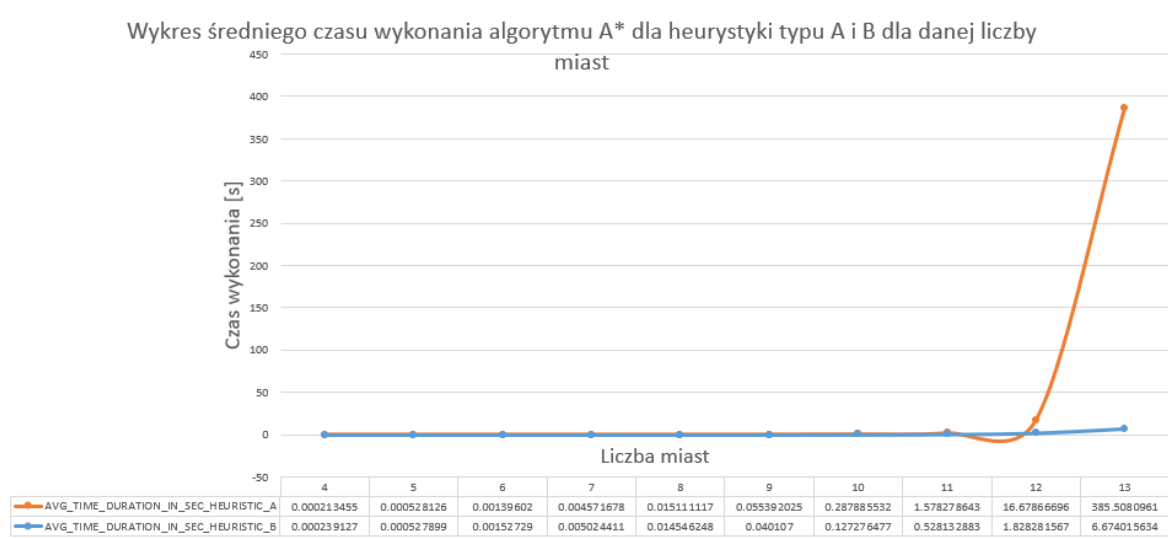


Rysunek 17 Wykres średniego i maksymalnego błędu względnego dla danej liczby miast i obu heurystyk dla algorytmu A*

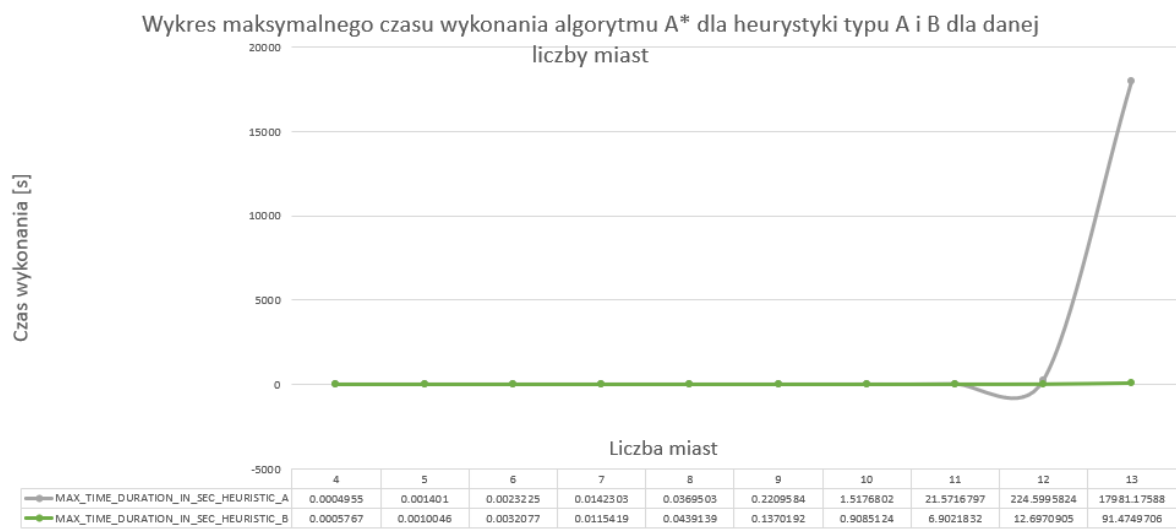
Powyższe dane obrazują, że wartość maksymalnego błędu względnego dla ilości miast większej niż 7 dla wykorzystania heurystyki typu B jest mniejsza od wykorzystania heurystyki A o blisko od 5 do 10 %. Wyniki zdają się wskazywać, że wykorzystanie heurystyki opartej o sumę niż iloczyn jest bardziej efektywne, ale by to potwierdzić należało jeszcze zbadać minimalny, średni oraz maksymalny zarejestrowany czas wykonania się algorytmu A* dla obu heurystyk i danej liczby miast, których wyniki zostały przedstawione na poniżej prezentowanych wykresach.



Rysunek 18 Wykres minimalnego czasu wykonania algorytmu A* dla danej liczby miast i obu badanych heurystyk



Rysunek 19 Wykres średniego czasu wykonania algorytmu A* dla danej liczby miast i obu badanych heurystyk

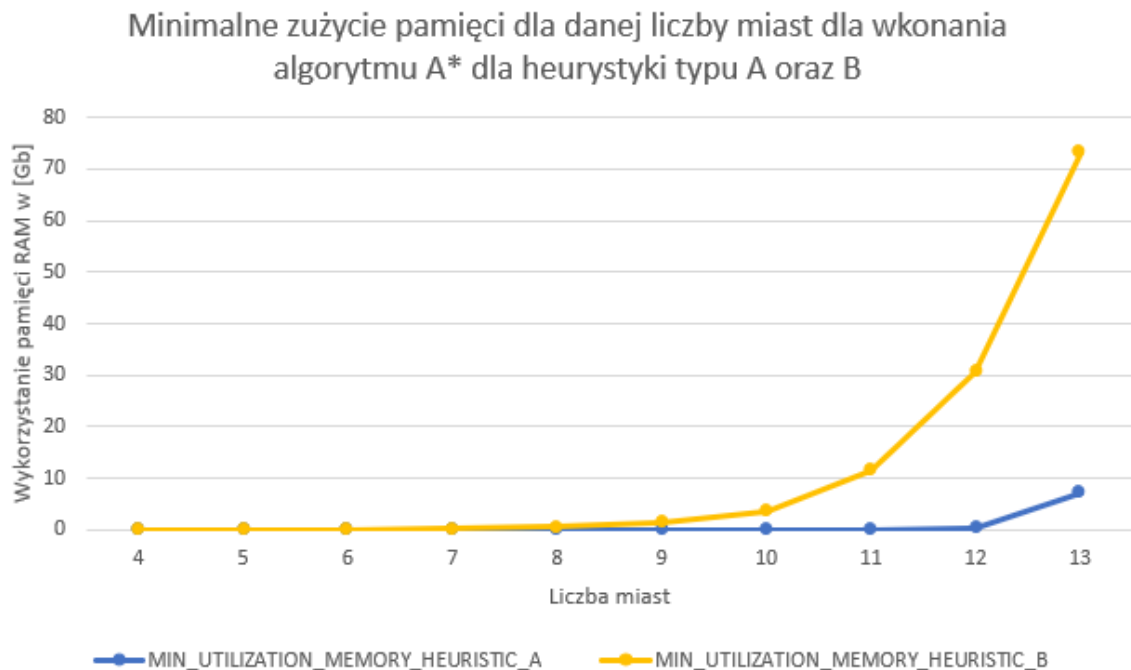


Rysunek 20 Wykres maksymalnego czasu wykonania algorytmu A* dla danej liczby miast i obu badanych heurystyk

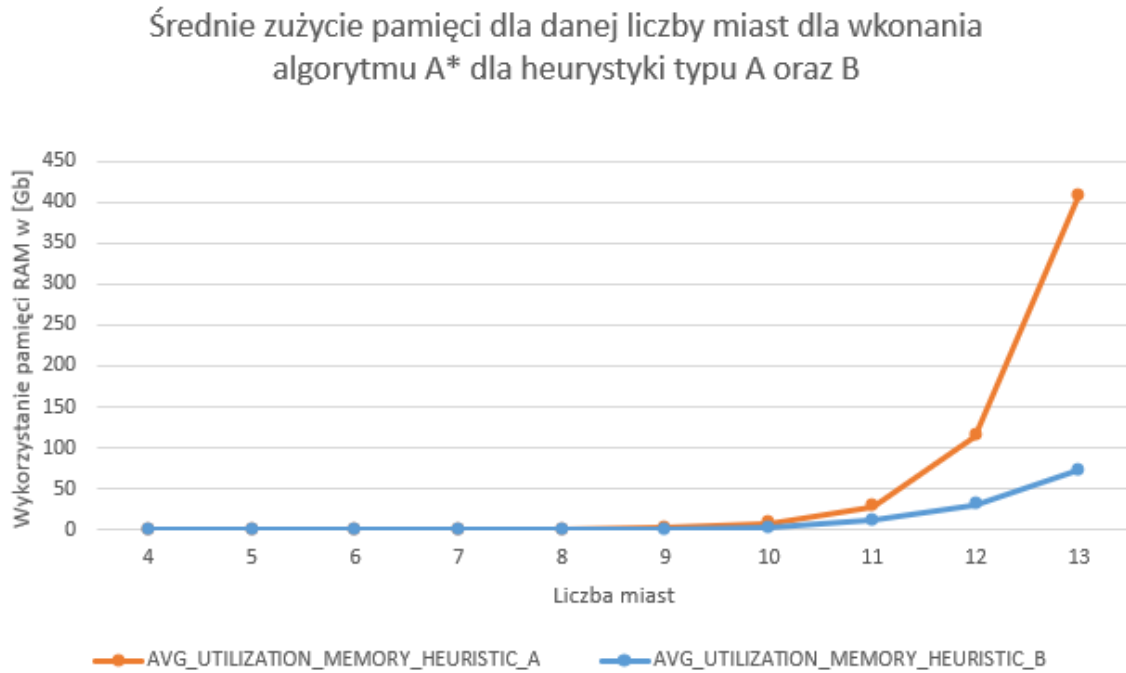
Analizując dane czasu wykonania algorytmu dla obu heurystyk, możemy dostrzec wyraźnie, że heurystyka A w problemie TSP dla liczby miast liczącej więcej niż 11 będzie znacznie dłużej poszukiwała rozwiązania problemu niż gdyby użyto heurystyki B.

7.2.3 A* analiza zużycia pamięci RAM dla obu heurystyk

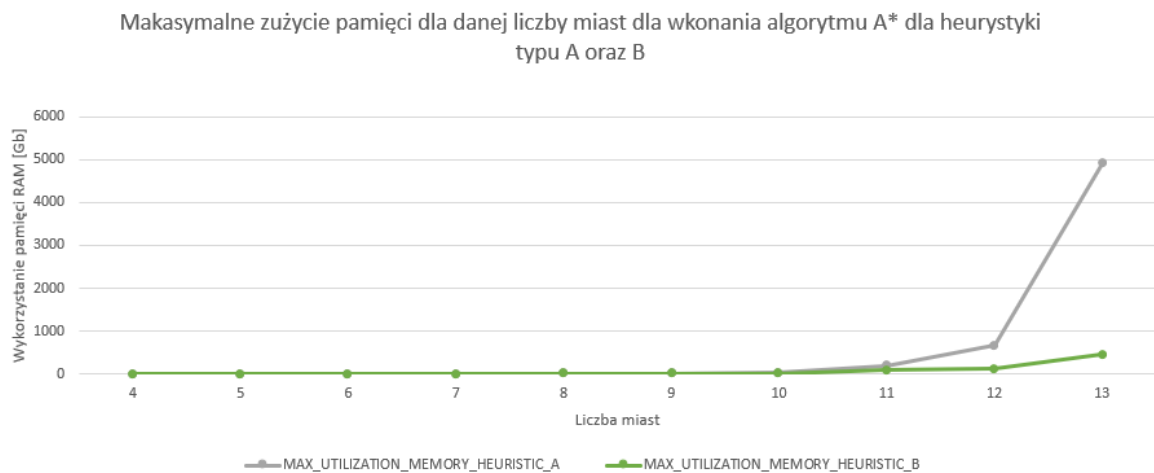
Ostatnią własnością jaką autor zbadał dla wykorzystania obu heurystyk, była ilość zaalokowanej przez nich pamięci, niezbędnej by móc wypracować bliskie optymalnego rozwiązanie zadanego symetrycznego problemu TSP. Poniżej zostały przedstawione wykresy przedstawiające minimalne, średnie oraz maksymalne zapotrzebowanie na pamięć zaimplementowanego przez autora algorytmu A*.



Rysunek 19 Minimalne zużycie pamięci zarejestrowane dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A*



Rysunek 20 Średnie zużycie pamięci w mierzone w gigabajtach dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A*



Rysunek 21 Maksymalne zużycie pamięci mierzone w gigabajtach dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A*

Dla implementacji algorytmu A* zastosowanej przez autora, kluczowym elementem było wykorzystanie kolejki priorytetowej zrealizowanej w ramach kopca. Rozwiązanie to pozwoliło zapewnić by dostęp do elementu o najniższym priorytecie (najmniejszej wartości funkcji $f(x) = g(x) + h(x)$) posiadał złożoność $O(1)$, oraz usunięcie elementu z kopca i przywrócenie kopcowi jego formę posiada złożoność $O(n \log n)$ natomiast operacja

Dodawanie nowego elementu do kopca charakteryzuje się w najgorszym przypadku złożonością $O(\log n)$. Dzięki wykorzystaniu struktury kopca a nie listy nie jest wymagane ciągłe wykonywanie operacji jej sortowania by znaleźć element o najniższym priorytecie. Jednakże oprócz kopca należy również zapewnić strukturę danych, która zabezpieczy nas przed osadzeniem w nim elementu który już się w nim znajduje, i tutaj potrzebna była do wykorzystania struktura danych o nazwie zbiór, wykorzystująca tablicę hashującą by złożoność sprawdzenia czy element zawiera już się w zbiorze posiadała złożoność $O(1)$.

Bazując na danych zobrazowanych na powyższych wykresach, można stwierdzić, że heurystyka B wraz ze wzrostem przestrzeni możliwych rozwiązań, pozwala na większą oszczędność pamięci niż wykorzystanie heurystyki A. Badanie zużycia pamięci również wykazały że istnieje pewien zbiór cech problemu TSP, który jest odpowiedzialny za tak dużą rozbieżność w między wartością minimalną, średnią oraz maksymalną czasu wykonania algorytmu A^* oraz jego zapotrzebowania na pamięć operacyjną w celu przechowywania wyników cząstkowych przybliżających do rozwiązania problemu TSP.

7.3 Analiza algorytmu przeszukiwań lokalnych

Jak zostało wcześniej powiedziane algorytm przeszukiwań lokalnych ma za zadanie modyfikować istniejące rozwiązanie przez z góry założoną skończoną ilość iteracji lub do czasu, gdy przez k iteracji nie udało się nam otrzymać rozwiązania lepszego. W ramach badań nad tą metaheurystyką postanowiłem zbadać czas wykonania, odległość uzyskanego rozwiązania od rozwiązania optymalnego oraz jak dużo pamięci RAM jest wymagane by rozwiązać tym sposobem symetryczny problem komiwojażera z użyciem biblioteki `tsp - python`. Do badań zostało wybranych 7 algorytmów perturbacji opisanych poniżej.

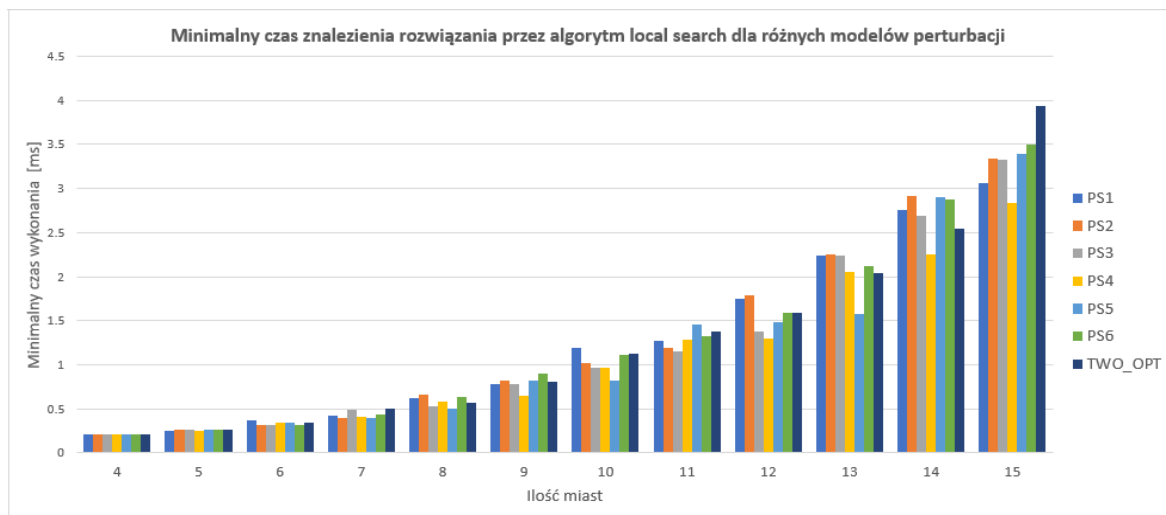
Badane algorytmy perturbacji:

- PS1 – zamiana dwóch sąsiadujących ze sobą elementów ($n-1$ zamian)
- PS2 – zamiana dowolnych dwóch elementów $(n*(n-1)) / 2$ zamian)
- PS3 – przeniesienie pojedynczego elementu ($n*(n-1)$ zamian)
- PS4 – przeniesienie podciągu
- PS5 – odwrócenie podciągu

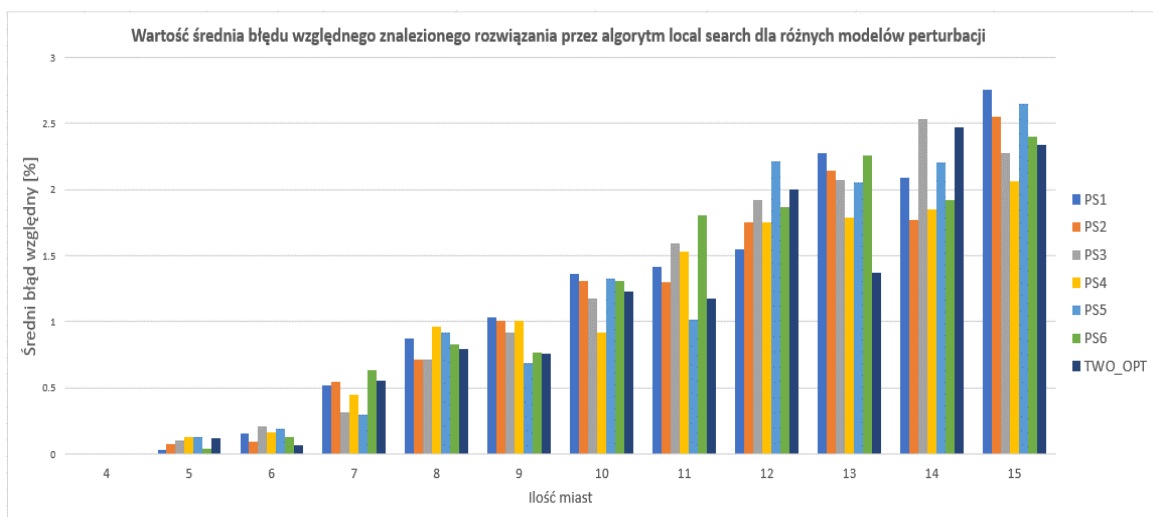
- PS6 – odwrócenie i przeniesienie podciągu
- TWO-OPT – wykorzystanie algorytmu 2-opt [27]

7.3.1 Analiza pomiarów efektywności i czasu wykonania

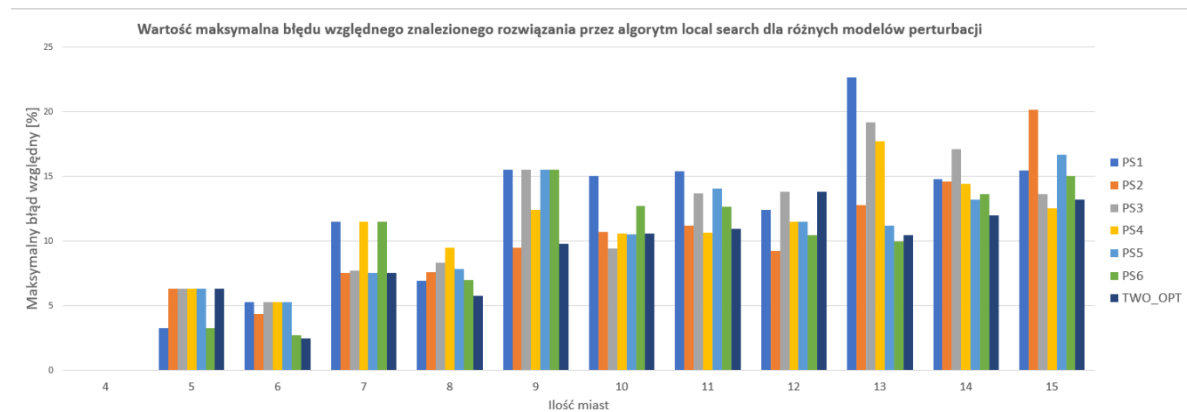
Poniżej prezentowane wyniki zostały zebrane przez wykonanie algorytmu local search na 1500 próbkach po 100 dla każdej liczby miast z przedziału od 4 do 15.



Rysunek 22 Wykres przedstawiający minimalny czas wykonania algorytmu local-search dla różnych modeli perturbacji



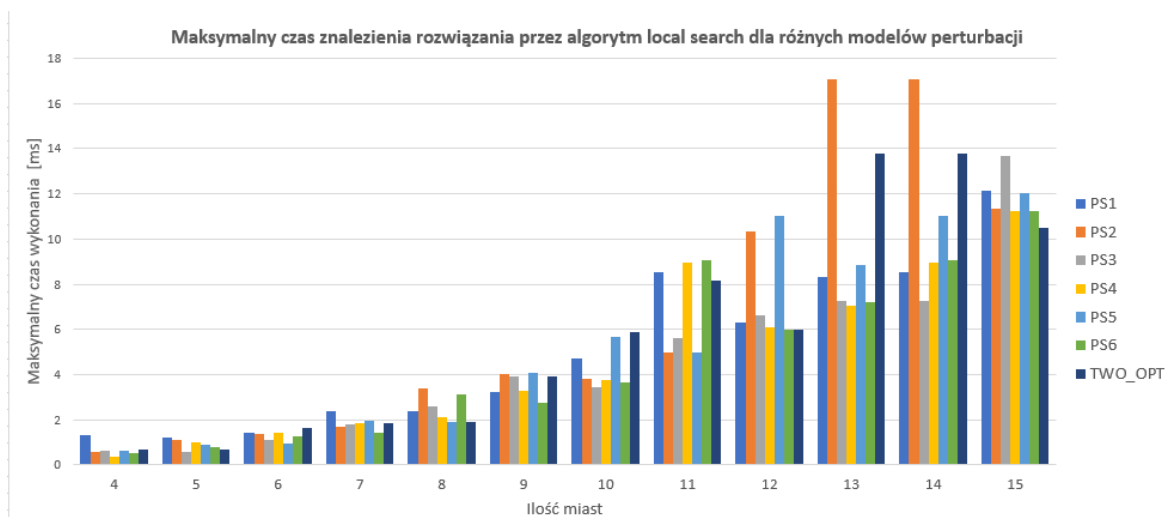
Rysunek 23 Wykres przedstawiający średni błąd względny dla rozwiązań uzyskanych z użyciem algorytmu local-search



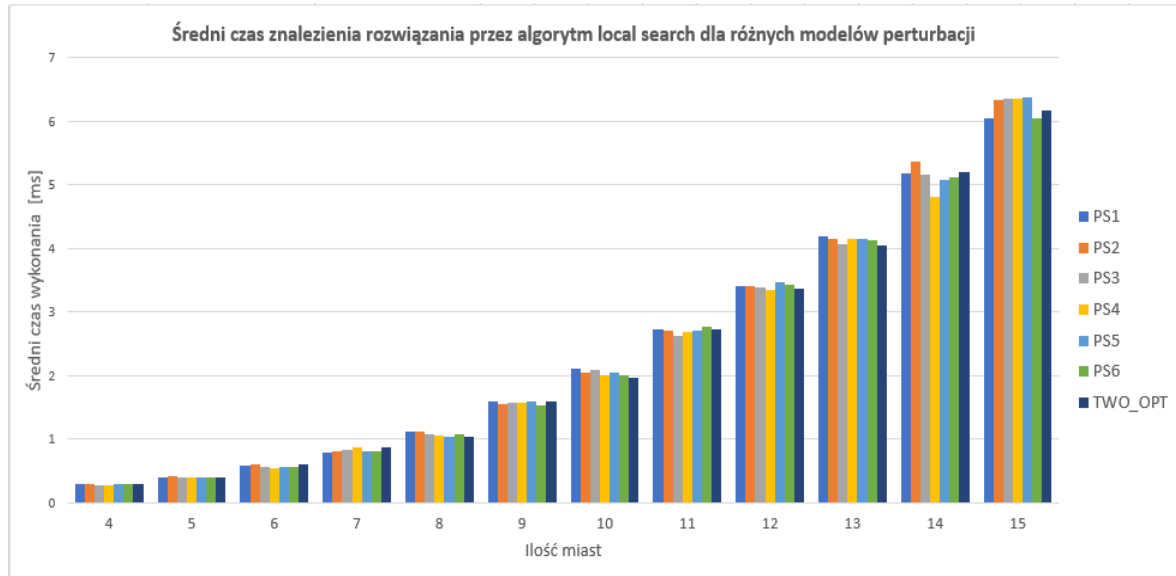
Rysunek 24 Wykres przedstawiający maksymalny błąd względny dla rozwiązań uzyskanych z użyciem algorytmu local-search

Na powyżej przedstawionych diagramach porównując średni i maksymalny błąd względny uzyskanych rozwiązań najlepsze wyniki efektywności odnotowano dla algorytmu „two-optimalnego”, którego średnia nieskuteczność była niższa niż 2,5% natomiast maksymalny błąd względny dla stale rosnącej liczby miast zawsze stanowił mniej niż 15% odległości od optymalnego rozwiązania.

Gdyby autor pracy musiałby wykorzystać algorytm local-search do znalezienia dosyć dobrego rozwiązania problemu TSP w komercyjnym projekcie, najpewniej wykorzystałby do tego algorytm two-opt, jednak by ustalić czy to był by najlepszy wybór z możliwych należy przyrzeć się również średniemu i maksymalnemu czasowi wykonania dla wszystkich modeli.



Rysunek 27 Wykres przedstawiający maksymalny czas przejścia algorytmu local-search dla różnych modeli perturbacji

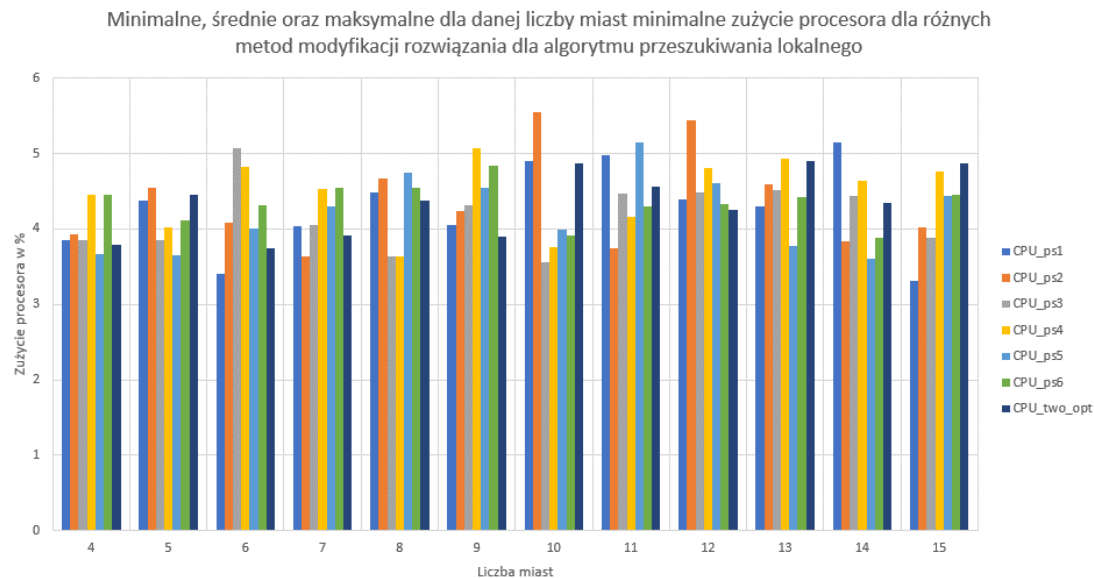


Rysunek 28 Wykres przedstawiający średni czas wykonania algorytmu local search dla danego modelu perturbacji

Powyżej zebrane dane przedstawiające czasy wykonania się algorytmów w milisekundach zdają się nadal potwierdzać, że najefektywniejszym modelem przekształcania rozwiązań jest algorytm „two-optymalny”, ponieważ dla 15 miast posiada najniższy maksymalny czas wypracowania rozwiązania oraz średni czas jego działania jest zbliżony do średnich czasów przejść innych modeli.

7.3.2 Analiza pomiarów zużycia CPU

Ze względu na prostotę działania algorytmu przeszukiwania lokalnego, autor pracy podejrzewa, że jego wykonanie nie powinno dokonać dużej konsumpcji CPU. Poniżej zostały przedstawione pomiary zużycia CPU wykonane na 1200 próbkach po 100 dla każdej liczby miast z przedziału od 4 do 15.



Rysunek 29 Wykres obrazujący minimalne zużycie procesora przez algorytm local-search dla różnych modeli perturbacji

Wyniki badań przeprowadzonych przez autora pracy wskazują, że wykonanie algorytmu przeszukiwań lokalnych nie wymaga dużego zużycia CPU. Zarejestrowane zużycie procesora oscyluje w granicach 3,5 – 5,5%. Minimalne, maksymalne oraz średnie zanotowane zużycie CPU jest identyczne ze względu na to, że wykonanie algorytmu zajęło mniej niż 1s a pomiary zużycia procesora były realizowane w 1s odstępach czasu, tzn., że na każde uruchomienie algorytmu dla danej próbki został wykonany jedynie pojedynczy pomiar zużycia CPU.

7.4 Analiza wyników algorytmu mrówkowego

W ramach przeprowadzonego szeregu badań dla zadanego zbioru problemów TSP o zadanej liczbie miast oraz znając dla wartość najlepszego ich rozwiązania, autor ma zamiar znaleźć optymalny zestaw parametrów dla ACO. Autor zamierza go ustalić, po to by potem móc porównać zużycie CPU, czas wykonania, efektywność uzyskanych rozwiązań oraz zapotrzebowanie algorytmu mrówkowego na pamięć operacyjną by porównać uzyskane wyniki z wynikami uzyskanymi dla innych algorytmów mających za zadanie rozwiązywać te same symetryczne problemy TSP.

Autor zamierza znaleźć optymalny zestaw parametrów dla algorytmu ACO zgodnie z poniższymi założeniami

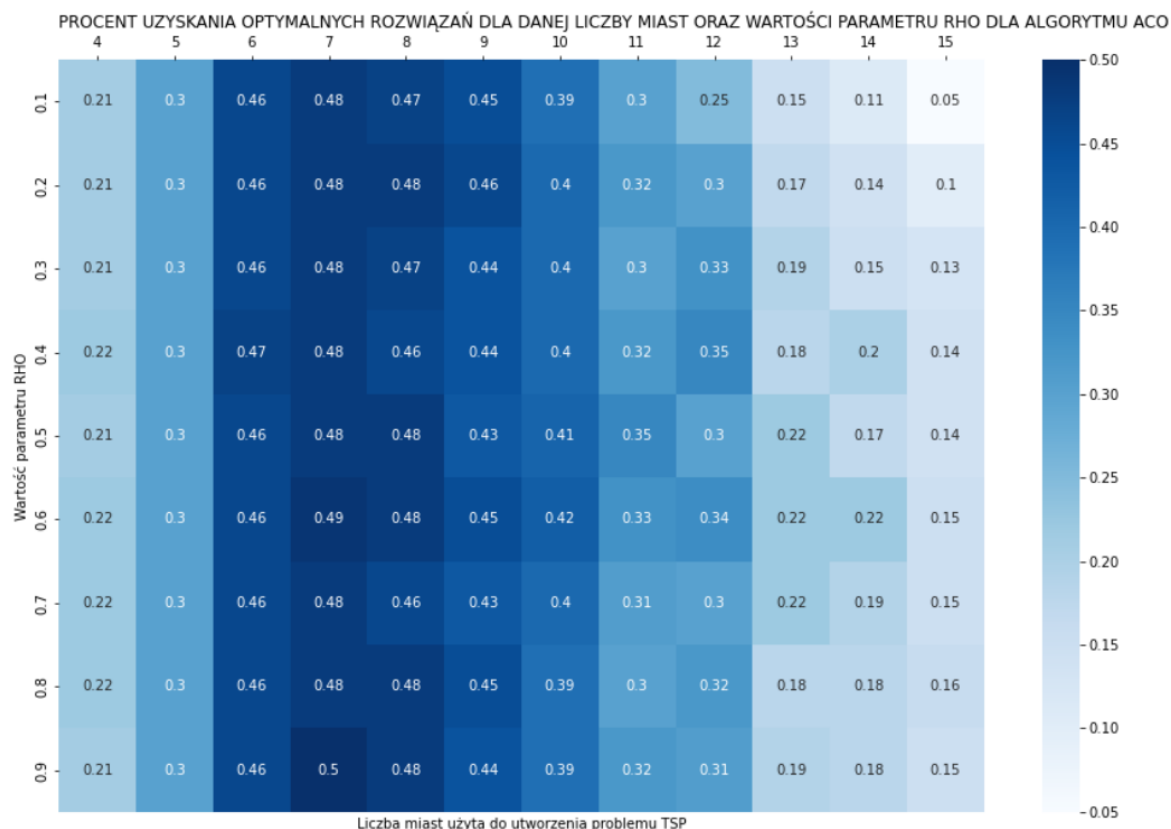
- maksymalna liczba iteracji: 20
- rozmiar populacji: 100
- wartość parametru alfa: 0.1 – 3.0 co 0.1
- wartość parametru beta: 0.1 – 3.0 co 0.1
- wartość parametru rho [0.1 – 0.9]

7.4.1 ACO poszukiwanie optymalnej wartości RHO

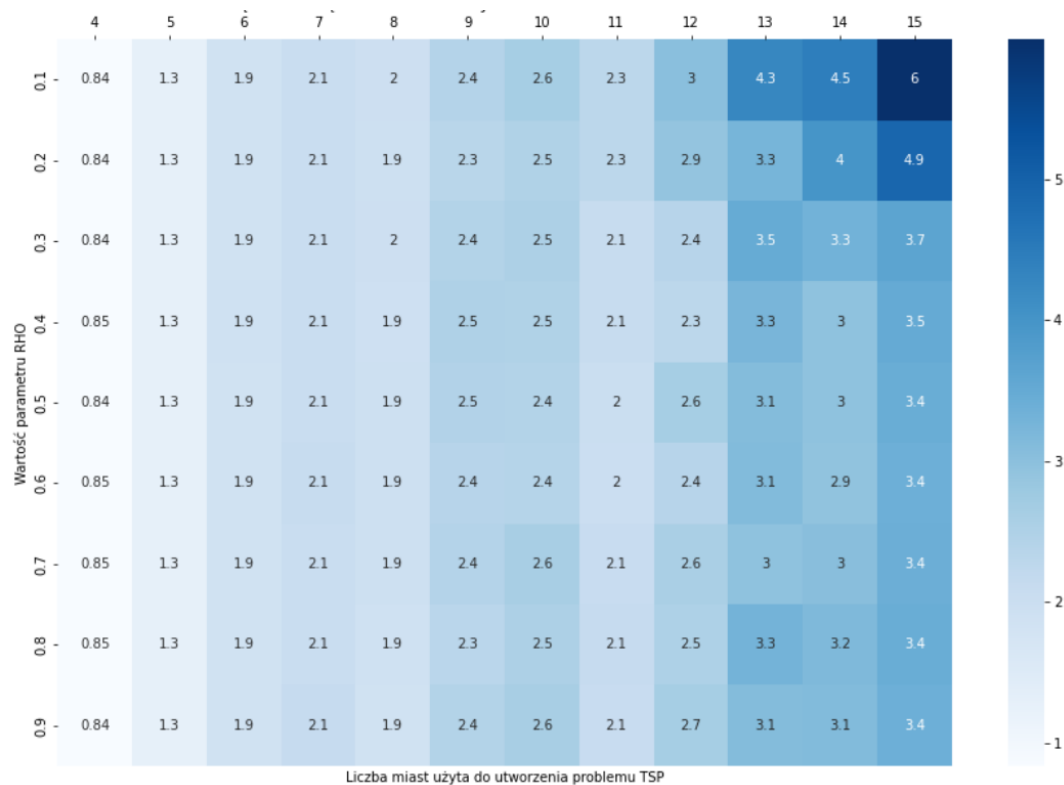
Do poszukiwania optymalnej wartości parametru RHO bazując na otrzymanych wartościach błędu względnego, postanowiono wykorzystać następujący zestaw stałych wartości parametrów.

- maksymalna liczba iteracji: 20 (domyślna wartość biblioteki scikit-opt [49])
- rozmiar populacji: 100
- wartość parametru alfa: 1 (domyślna wartość biblioteki scikit-opt [49])
- wartość parametru beta: 2 (domyślna wartość biblioteki scikit-opt [49])

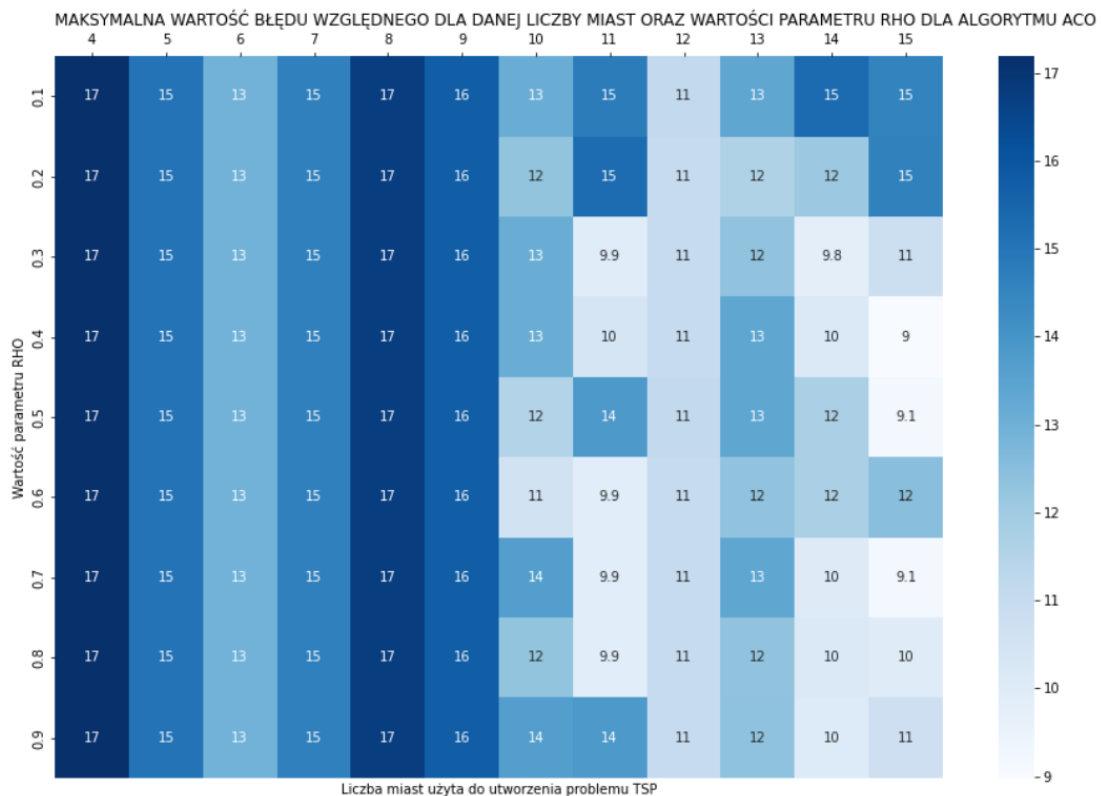
Bazując na powyższych założeniach, powstały poniższe mapy ciepła, które umożliwią na podstawie wybrania parametru RHO dla którego średnia oraz maksymalna wartość błędu względnego otrzymanych rozwiązań dla zadanej liczby miast rośnie najwolniej. Do badań na daną liczbę miast przypadło 100 próbek, umożliwiających otrzymanie obiektywnych wyników.



Rysunek 31 Mapa cieplna obrazująca procent odnalezienia optymalnego rozwiązania dla danej liczby miast oraz danej wartości parametru RHO dla algorytmu ACO



Rysunek 30 Mapa cieplna średniej wartości błędu względnego dla danej wartości RHO i liczby miast



Rysunek 32 Mapa cieplna maksymalnej wartości błędu względnego dla danej wartości RHO i liczby miast

Bazując na wynikach powyżej przedstawionych map cieplnych można wywnioskować, że optymalna wartość parametru RHO znajduje się pomiędzy wartościami 0.6 oraz 0.8. W konsekwencji pozyskanych wniosków do dalszych badań z użyciem algorytmu mrówkowego zostanie wykorzystana wartość parametru RHO = 0.7.

7.4.2 ACO poszukiwanie wartości parametrów alfa i beta

//TODO

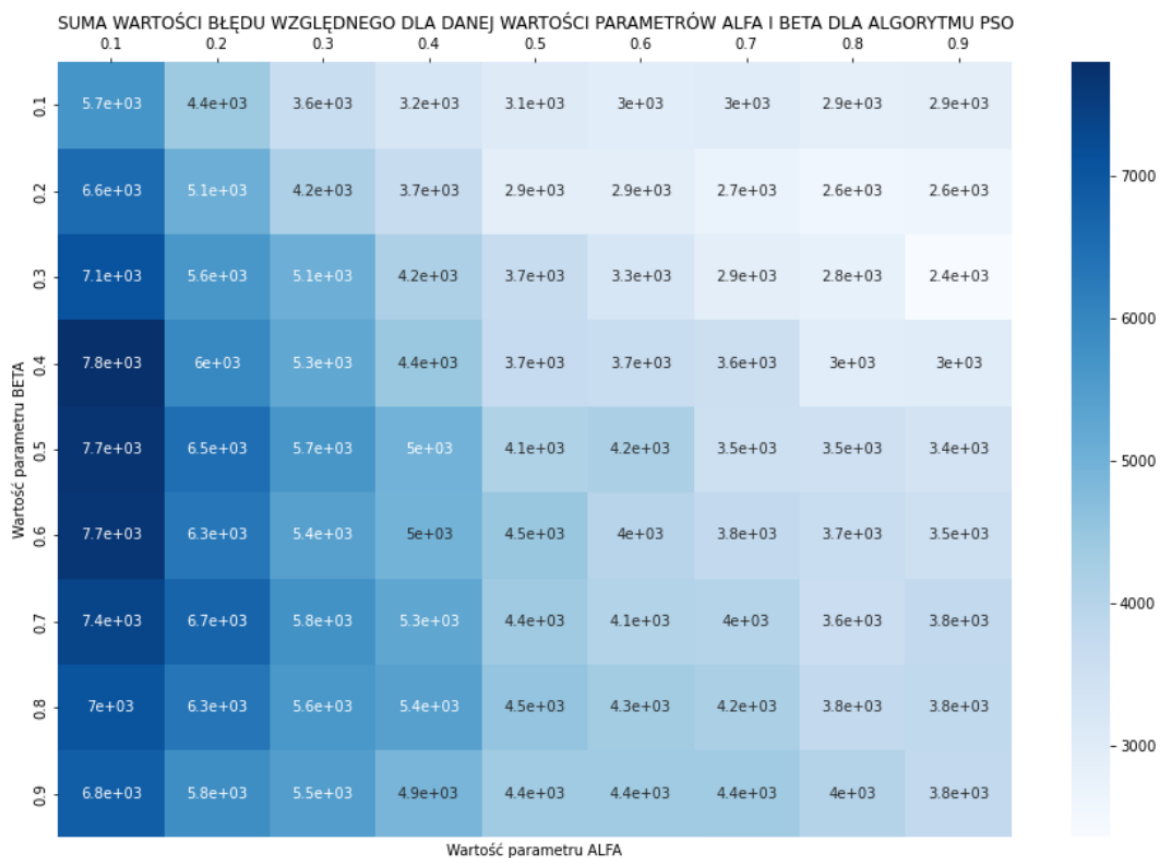
7.5 Analiza wyników algorytmu roju cząstek (PSO)

Algorytm roju cząstek, stanowi jeden z ciekawszych rodzajów algorytmów heurystycznych, gdyż jego domyślna implementacja ma zastosowanie do poszukiwania ekstremum funkcji wielu zmiennych. W przypadku gdy postanowiliśmy dostosować jego działania do rozwiązywania problemu TSP należało upodobnić jego działanie do działania algorytmu genetycznego (zastosowanie operatora jedno genowego SWAP do modyfikacji rozwiązania danej cząstki) by z odpowiednim prawdopodobieństwem α cząstka modyfikowała swoje rozwiązanie na wzór wcześniej najlepszego przez nią znalezione

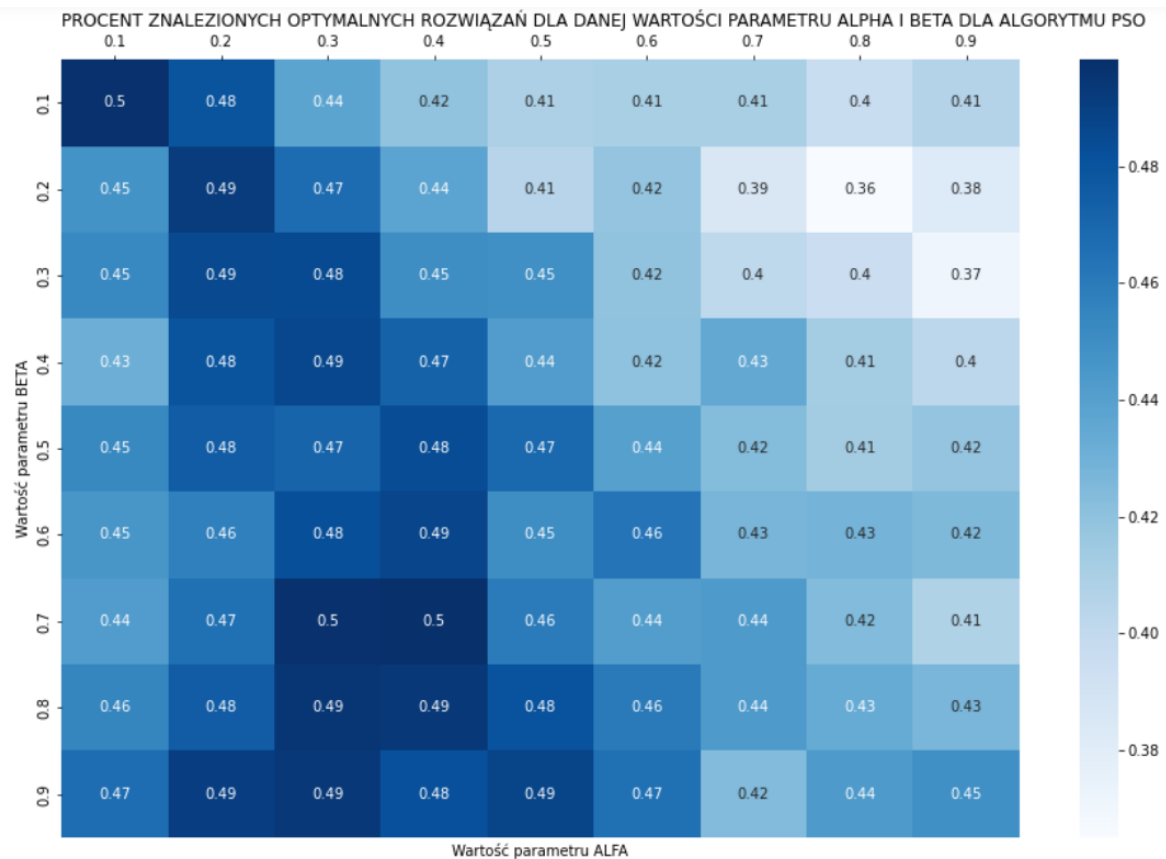
oraz by z prawdopodobieństwem β dążyła do ulepszenia własnego rozwiązania na wzór najlepszego znalezionej przez cały rój. Autor niniejszej pracy postawił sobie za cel zbadanie dla jakich wartości prawdopodobieństw α oraz β algorytm PSO okaże się najbardziej skuteczny do rozwiązywania problemu TSP przy założeniu, że dysponujemy liczbą 100 cząstek i wykonamy 100 iteracji algorytmu. Jak w przypadku innych badań do tych również została wykorzystana pula 1200 próbek problemów TSP podzielonych po 100 próbek na każdą liczbę miast z przedziału od 4 do 15.

7.5.1 PSO poszukiwanie wartości parametrów alfa oraz beta

Poszukiwania optymalnych wartości parametrów alfa oraz beta autor pracy postanowił rozpocząć od określenia wartości procentowej znalezienia optymalnych rozwiązań rozpatrywanych problemów TSP dla danych wartości alfa i beta z przedziału 0.1- 0.9 a następnie porównać która para parametrów niezależnie od badanej liczby miast pozwoliła osiągnąć najniższą sumę błędów względnych uzyskanych rozwiązań.



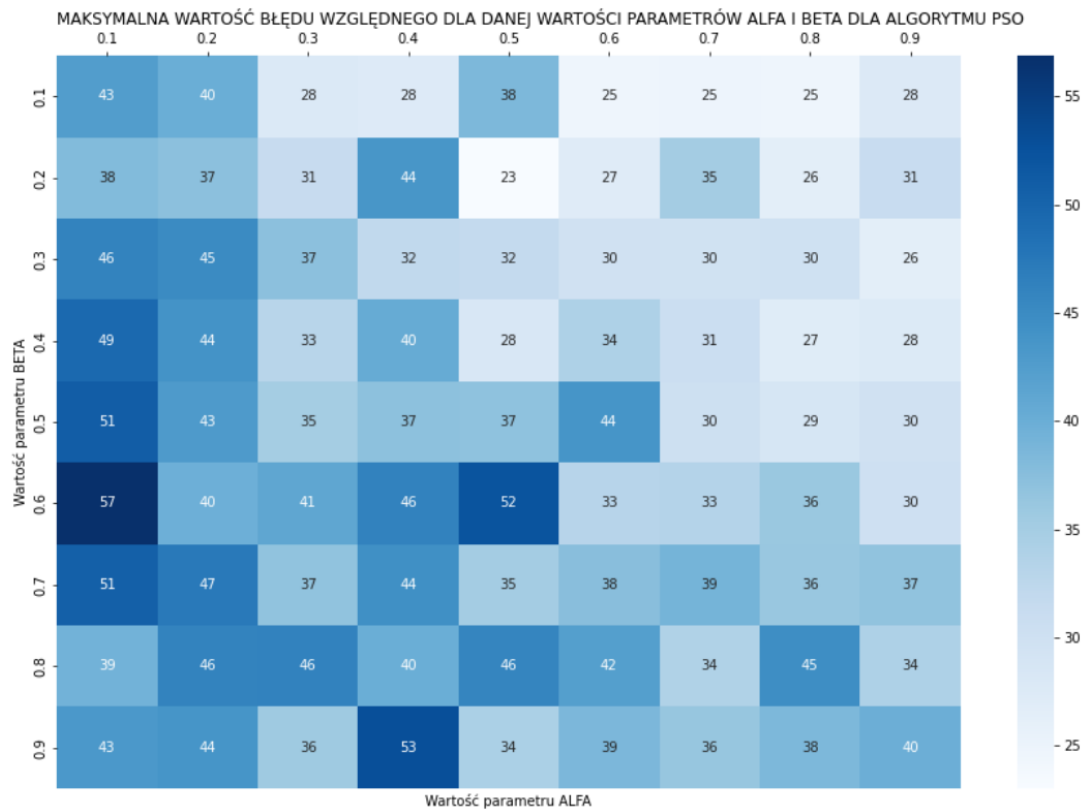
Rysunek 33 Mapa cieplna przedstawiająca procent znalezienia najlepszych rozwiązań dla 1500 problemów TSP dla danej wartości parametru alfa i beta z użyciem algorytmu PSO



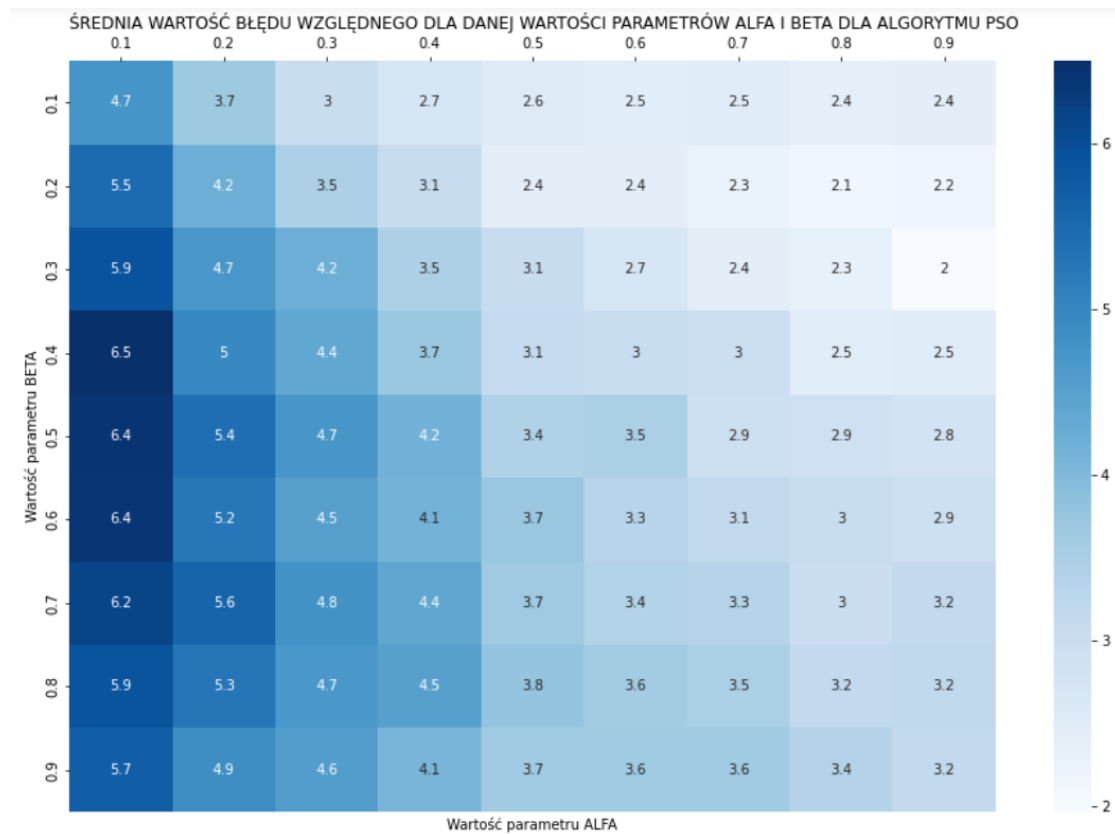
Rysunek 34 Mapa cieplna prezentująca sumę wartości błędów względnych dla badanej wartości parametrów alfa i beta

Na podstawie powyżej przedstawionych danych można zauważyć, że algorytm PSO największą liczbę optymalnych rozwiązań znalazł dla dwóch par parametrów alfa i beta tj. (alfa: 0.3; beta: 0.7 oraz alfa: 0.4, beta: 0.7) wtedy algorytm PSO znalazł około 50% wszystkich możliwych najlepszych rozwiązań symetrycznego problemu TSP dla badanych problemów liczących od 4 do 15 miast po 100 unikalnych problemów na każdą liczbę miast. Aby mieć pewność, że faktycznie te dwie wartości parametrów algorytmu PSO są poprawne należy dodatkowo weryfikować czy i dla nich suma wartości względnych błędów będzie również najniższa.

Wyniki analizy sumy wartości błędów względnych pokazały, że wartości parametru alfa i beta wskazane na podstawie analizy poprzedniej mapy cieplnej nie pokrywają się. Oznacza to, że algorytm PSO można skonfigurować na dwa sposoby albo by zmaksymalizować szansę osiągnięcia wyniku optymalnego przez ustawienie parametrów alfa i beta na wartości 0.3-0.4 i 0.7 lub by zmaksymalizować szansę na osiągnięcie wyników najbliższych optymalnych tj. z najmniejszym osiągniętym błędem względnym dla parametru alfa i beta równym 0.9 i 0.3. Aby potwierdzić która para parametrów jest poprawna należy przyrzeć się również zależności średniej i maksymalnej wartości błędu względnego zależnej od wartości obu parametrów.



Rysunek 35 Mapa cieplna prezentująca maksymalną wartość błędu względnego dla badanych wartości parametrów alfa i beta



Rysunek 36 Mapa cieplna prezentująca średnią wartość błędu względnego dla badanych wartości parametrów alfa i beta

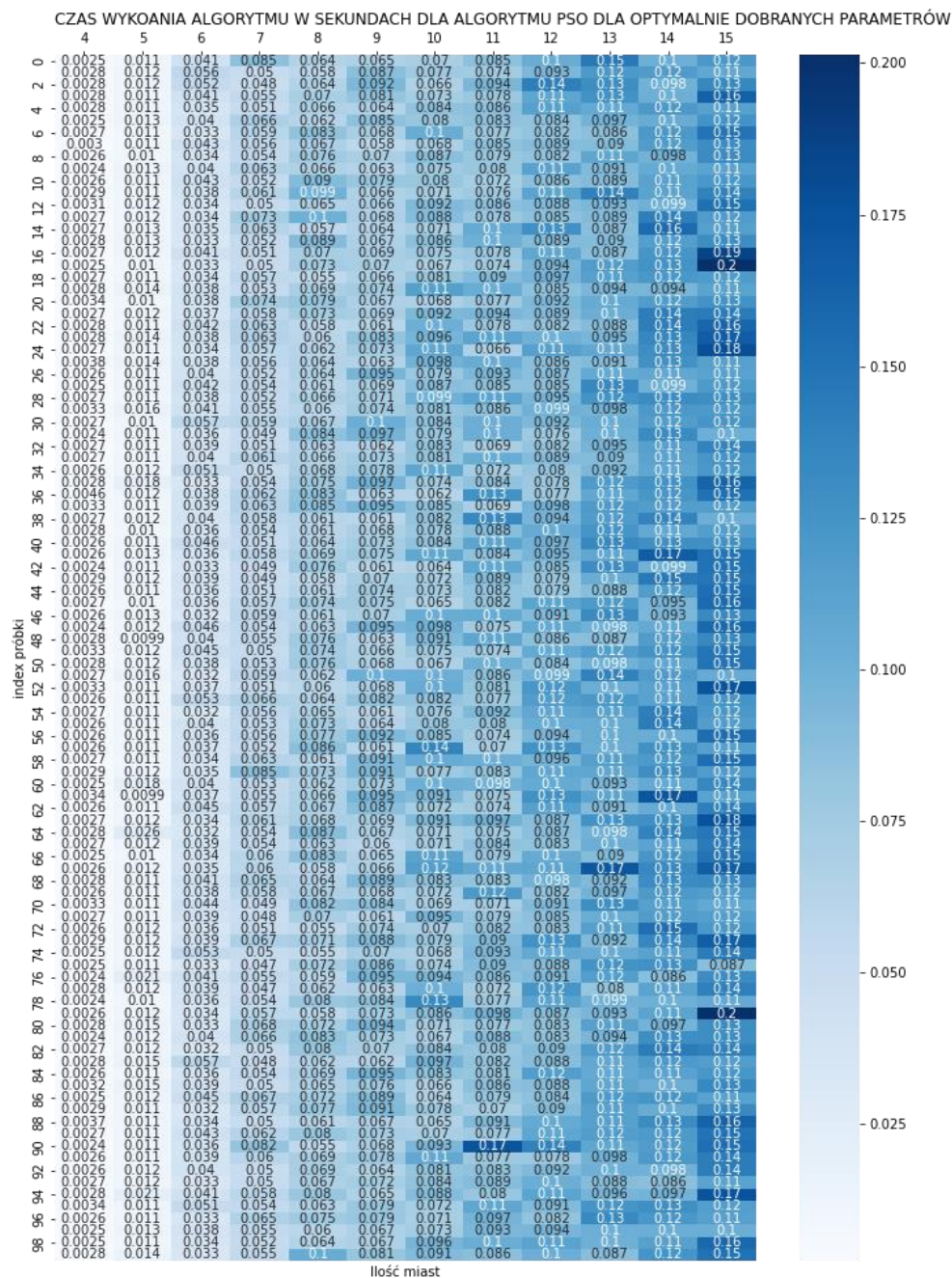
Najniższa średnia wartość błędu względnego również została wykryta dla pomiarów które zostały przeprowadzone przez algorytm PSO z wartościami parametrów alfa i beta wynoszącymi 0.9 oraz 0.3 i jest równa 2%. Dla tych samych wartości parametrów została zanotowana nie najniższa, ale dosyć mała wartość maksymalnego błędu względnego wynosząca blisko 26% gdzie najniższa wartość tego samego parametru jest równa 23%. Bazując na powyżej przedstawionych danych autor postanowił uznać, że najlepszym dopasowaniem parametrów z użyciem algorytmu PSO niezależnie od stopnia skomplikowania problemu, ilości iteracji oraz populacji cząstek jest wartość parametru alfa równa 0.9 oraz wartość parametru beta wynosząca 0.3 i taki zakres parametrów zostanie użyty do przeprowadzenia dalszych badań z użyciem tego algorytmu.

7.5.2 PSO analiza czasu wykonania dla wybranych parametrów

W poprzednim rozdziale zostały wyznaczone parametry alfa i beta, które zapewniają najlepsze rozwiązania problemu TSP. W tym rozdziale autor postanowił się bliżej przyjrzeć czasu wykonania algorytmu PSO dla wcześniej wyznaczonej liczby miast oraz próbek dla poniżej wybranych wartości parametru algorytmu.

Parametru algorytmu PSO dla których zostały poniżej przedstawione poniższe mapy ciepłe obrazujące czas wykonania algorytmu.

- Liczba iteracji: 100
- Rozmiar populacji: 100
- Wartość parametru alpha (prawdopodobieństwo zaufania cząstki w znalezione przez siebie najlepsze rozwiązanie problemu): 0.9
- Wartość parametru beta (prawdopodobieństwo zaufania cząstki w znalezione przez rój najlepsze rozwiązanie problemu): 0.3



Rysunek 35 Mapa cieplna prezentująca czas wykonania algorytmu PSO dla optymalnej wartości parametrów α i β dla danej liczby miast i indexu badanej próbki

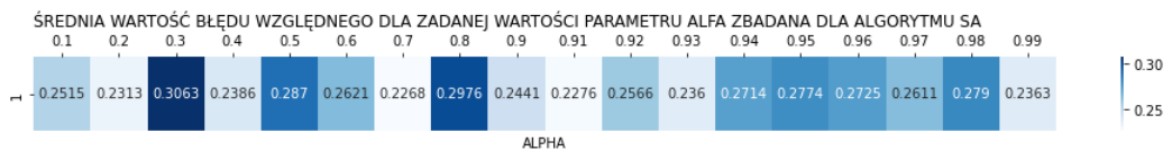
Powyżej zaprezentowane wyniki, informują nas, że pomimo ustalenia stałej liczby populacji cząstek i liczby iteracji to czas rozwiązania problemu TSP rośnie. Czas wykonania algorytmu wraz ze wzrostem przestrzeni problemu ulega coraz większym fluktuacjom, jest to spodziewane zachowanie, ponieważ algorytm PSO należy do grupy algorytmów stochastycznych i decyzja o wykonaniu modyfikacji rozwiązania zachodzi w sposób losowy.

7.6 Analiza wyników symulowanego wyżarzania (SA)

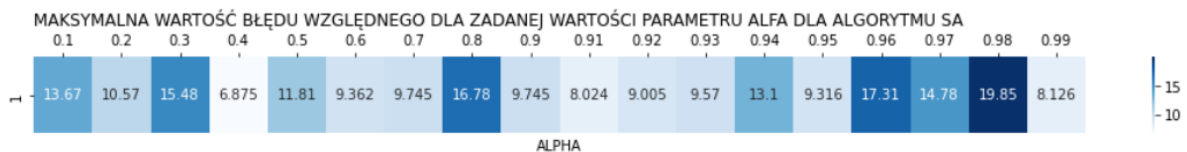
Implementacja algorytmu symulowanego wyżarzania z biblioteki `python_tsp` pozwala użytkownikom dostosować 2 parametry tj. schemat perturbacji (modyfikowania) oraz parametr alfa (wartość $0 < \text{alfa} < 1$) stanowiący o szybkości spadku temperatury im wartość alfa jest niższa tym spadek temperatury postępuje szybciej.

7.6.1 SA, wartości parametru alfa a błąd względny

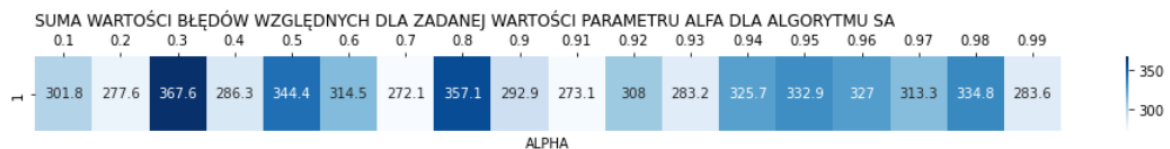
W ramach przeprowadzonych badań autor pracy zastosował algorytm SA dla 1200 przypadków symetrycznego problemu TSP modyfikując otrzymane rozwiązania z użyciem algorytmu „dwu-optimalnego” [27] oraz wartość parametru alfa z zakresu $[0.1 - 0.9] \cup [0.91 - 0.99]$. Autor zamierza porównać średnie i maksymalne wartości błędów względnych oraz wartość procentową znalezienia optymalnych rozwiązań dla danej wartości parametru alfa w celu optymalnej jego wartości.



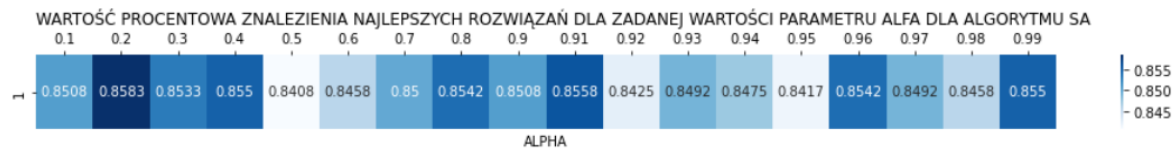
Rysunek 36 Mapa cieplna przedstawiająca średnią wartość błędu względnego dla danej wartości parametru alfa dla algorytmu SA



Rysunek 37 Mapa cieplna przedstawiająca maksymalną zarejestrowaną wartość błędu względnego dla danej wartości parametru alfa dla algorytmu SA



Rysunek 38 Mapa cieplna przedstawiająca sumę wartości błędów względnych dla danej wartości parametru alfa dla algorytmu SA



Rysunek 39 Mapa cieplna przedstawiająca wartość procentową znalezienia najlepszych rozwiązań dla zadanej wartości parametru alfa dla algorytmu SA

Bazując na danych przedstawionych na powyższych wykresach trudno jednoznacznie wyznaczyć optymalną wartość parametru alfa, z tego względu wnioski dotyczące każdego z powyższych wykresów autor pracy umieść w poniższej tabeli.

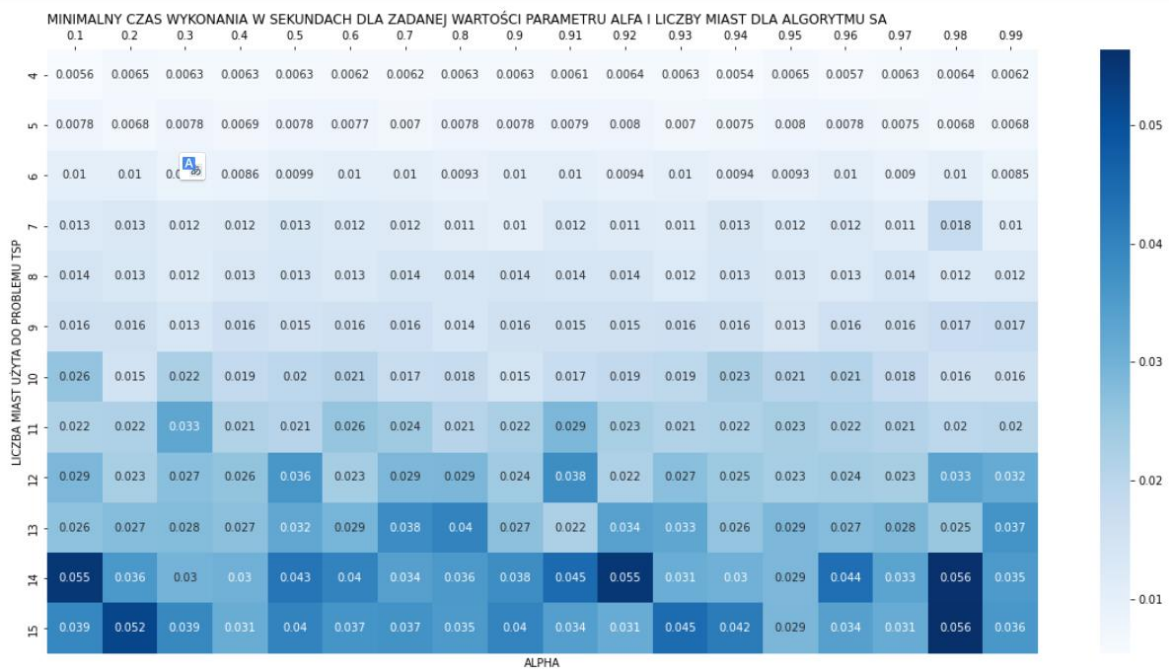
Tabela 3 Tabela przedstawiająca poszukiwanie optymalnej wartości parametru alfa dla algorytmu SA

Badana własność	alfa ekstremum nr 1.	alfa ekstremum nr 2.	alfa ekstremum nr 3.
minimum dla średnia wartości błędu względnego	0.8	0.91	0.2
minimum dla maksymalnej wartości błędu względnego	0.4	0.91	0.99
minimum dla sumy wartości błędów względnych	0.7	0.91	0.2
maksimum znalezienia optymalnej ilości rozwiązań wyrażonej w %	0.2	0.91	0.96

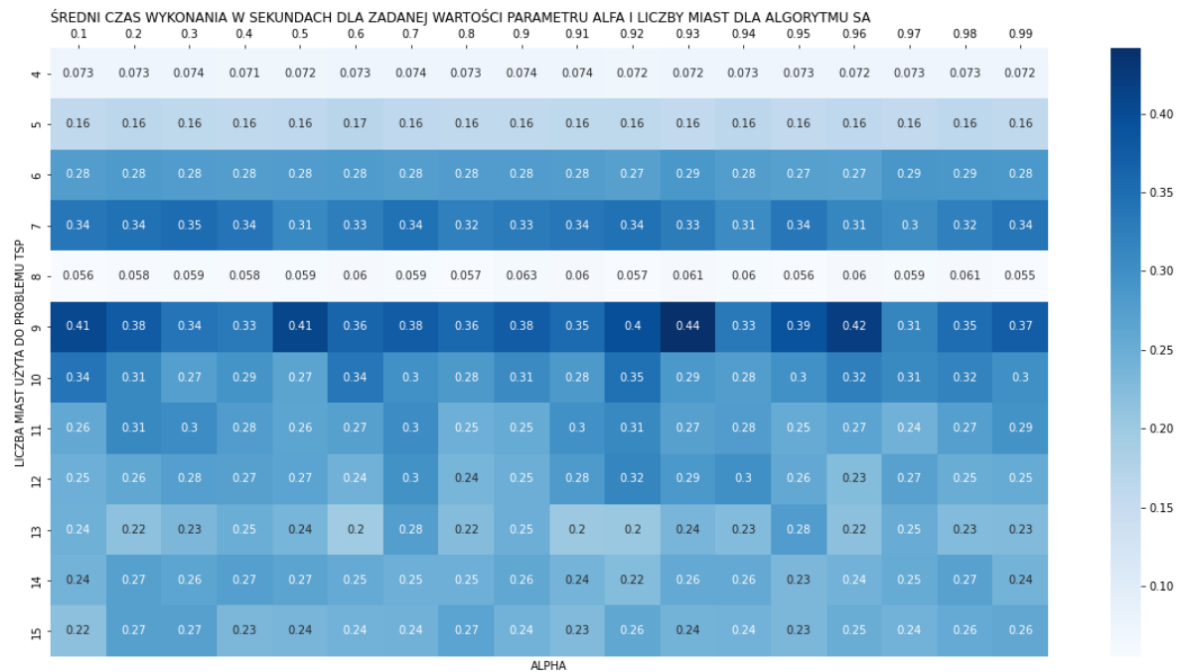
Autor pracy w powyższej tabeli uwzględnił porównanie wartości parametrów alfa zarejestrowane dla 3 ekstremów badanych własności. Zauważono, że wartość parametru alfa równa **0.91** powtarza się najczęściej w powyższym zestawieniu zatem zdaniem autora ta wartość powinna zostać uznana za optymalną.

7.6.2 SA, wartości parametru alfa a czas wykonania

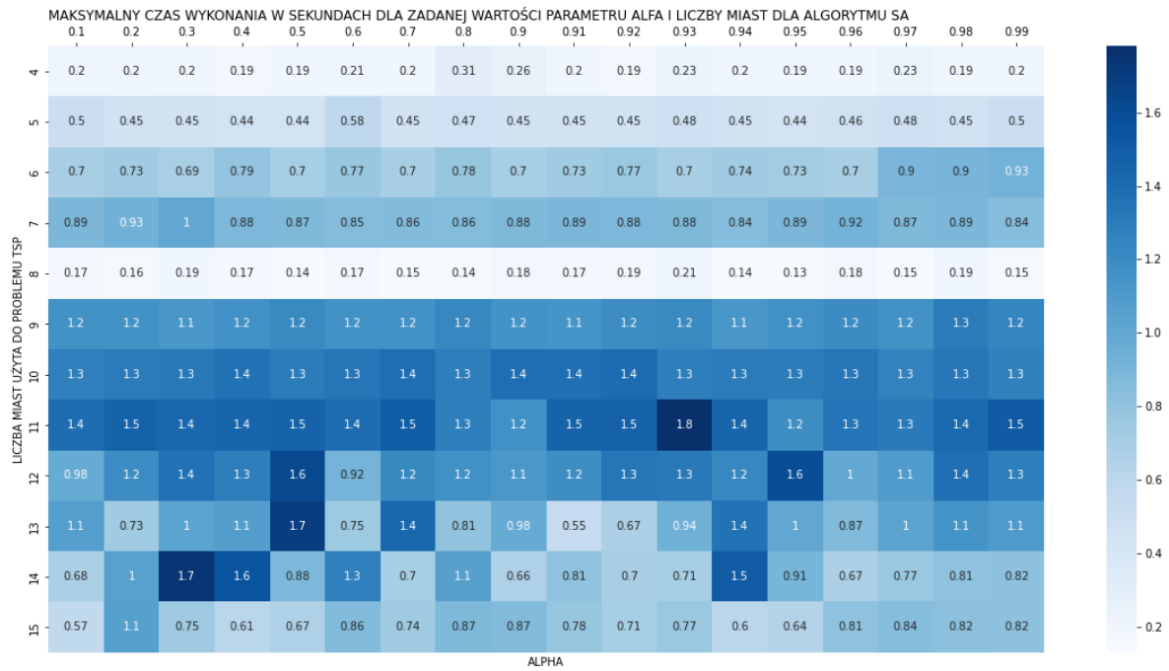
Do pełnego obrazu badanego algorytmu wymagane było porównanie minimalnego, średniego oraz maksymalnego czasu wykonania algorytmu SA, których wyniki są zależne od ilości miast użytych dla problemu TSP.



Rysunek 40 Mapa ciepła minimalnego czasu wykonania algorytmu SA w sekundach dla danej wartości parametru alfa i liczby miast



Rysunek 41 Mapa ciepła średniego czasu wykonania algorytmu SA w sekundach dla danej wartości parametru alfa i liczby miast



Rysunek 42 Mapa cieplna maksymalnego czasu wykonania algorytmu SA w sekundach dla danej wartości parametru alfa i liczby miast

Powyższe mapy cieplne pokazują, że czas wykonania algorytmu dla problemów TSP liczących do 15 miast zazwyczaj nie trwa więcej jak 1.8 sekundy, czyli sam problem komiwojażera z użyciem tego algorytmu jest rozwiązywany dość dokładnie a zarazem szybko, co faworyzuje go spośród innych badanych do referowania jego użycia do rozwiązywania problemów TSP.

7.6.3 SA, wartości parametru alfa a zużycie procesora

7.6.4 SA, wartości parametru alfa a zużycie pamięci RAM

7.7 Analiza wyników algorytmów genetycznych

//TODO

8. Konkluzje

//TODO

9. Spis ilustracji

Rysunek 1 Zależność problemów P, NP, NP-zupełnych i NP-trudnych [53]	7
Rysunek 2 Algorytm A* w pseudokodzie [28]	22
Rysunek 3 Fragment kodu algorytmu genetycznego w bibliotece mlrose realizujący operację krzyżowania	30
Rysunek 4 Fragment kodu realizujący mutację wstawiającą w AG z biblioteki mlrose	30
Rysunek 5 Fragment implementacji PSO dla TSP reprezentujący tworzenie 1 części wektora prędkości	37
Rysunek 6 Fragment implementacji PSO dla TSP definiujący 2 część wektora prędkości oraz tworzący nowe lepsze rozwiązanie na bazie wiedzy cząstki i roju	38
Rysunek 7 Schemat architektury pomiarowej	43
Rysunek 8 Kod źródłowy klasy CpuProfiler	44
Rysunek 9 Fragment kodu źródłowego odpowiedzialnego za pomiar zużycia pamięci z użyciem biblioteki trace malloc	45
Rysunek 10 Wykres obrazujący średni i minimalny błąd uzyskanych rozwiązań z użyciem algorytmu NN	46
Rysunek 11 Wykres minimalnego, średniego i maksymalnego czasu znalezienia rozwiązania TSP przez algorytm NN	47
Rysunek 12 Wykres średniego i maksymalnego zużycia CPU przez algorytm NN	48
Rysunek 13 Wykres minimalnej, średniej oraz maksymalnej wartości zarejestrowanego minimalnego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk	49
Rysunek 14 Wykres minimalnej, średniej i maksymalnej wartości zarejestrowanego średniego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk	50
Rysunek 15 Wykres minimalnej, średniej i maksymalnej wartości zarejestrowanego maksymalnego zużycia procesora dla danej liczby miast dla algorytmu A* dla różnych heurystyk	50

Rysunek 16 Wykres ilości uzyskanych rozwiązań optymalnych do błędnych dla danej liczby miast i obu badanych heurystyk 51

Rysunek 17 Wykres średniego i maksymalnego błędu względnego dla danej liczby miast i obu heurystyk dla algorytmu A^* 52

Rysunek 18 Wykres minimalnego czasu wykonania algorytmu A^* dla danej liczby miast i obu badanych heurystyk 53

Rysunek 19 Minimalne zużycie pamięci zarejestrowane dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A^* 54

Rysunek 20 Średnie zużycie pamięci mierzone w gigabajtach dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A^* 55

Rysunek 21 Maksymalne zużycie pamięci mierzone w gigabajtach dla danej liczby miast oraz obu heurystyk użytych do wykonania algorytmu A^* 55

Rysunek 22 Wykres przedstawiający minimalny czas wykonania algorytmu local-search dla różnych modeli perturbacji 57

Rysunek 23 Wykres przedstawiający średni błąd względny dla rozwiązań uzyskanych z użyciem algorytmu local-search 57

Rysunek 24 Wykres przedstawiający maksymalny błąd względny dla rozwiązań uzyskanych z użyciem algorytmu local-search 58

Rysunek 25 Wykres przedstawiający minimalny czas wykonania algorytmu local search dla różnych modeli perturbacji 58

Rysunek 26 Wykres przedstawiający minimalny czas wykonania algorytmu local search dla różnych modeli perturbacji 58

Rysunek 27 Wykres przedstawiający maksymalny czas przejścia algorytmu local-search dla różnych modeli perturbacji 58

Rysunek 28 Wykres przedstawiający średni czas wykonania algorytmu local search dla danego modelu perturbacji 59

Rysunek 29 Wykres obrazujący minimalne zużycie procesora przez algorytm local-search dla różnych modeli perturbacji 60

Rysunek 30 Mapa cieplna obrazująca procent odnalezienia optymalnego rozwiązania dla danej liczby miast oraz danej wartości parametru RHO dla algorytmu ACO 62

Rysunek 31 Mapa cieplna średniej wartości błędu względnego dla danej wartości RHO i liczby miast 62

Rysunek 32 Mapa cieplna maksymalnej wartości błędu względnego dla danej wartości RHO i liczby miast 63

Rysunek 33 Mapa cieplna przedstawiająca procent znalezienia najlepszych rozwiązań dla 1500 problemów TSP dla danej wartości parametru alfa i beta z użyciem algorytmu PSO 64

Rysunek 34 Mapa cieplna prezentująca sumę wartości błędów względnych dla badanej wartości parametrów alfa i beta 65

Rysunek 35 Mapa cieplna prezentująca czas wykonania algorytmu PSO dla optymalnej wartości parametrów alfa i beta dla danej liczby miast i indexu badanej próbki 69

Rysunek 36 Mapa cieplna przedstawiająca średnią wartość błędu względnego dla danej wartości parametru alfa dla algorytmu SA 70

Rysunek 37 Mapa cieplna przedstawiająca maksymalną zarejestrowaną wartość błędu względnego dla danej wartości parametru alfa dla algorytmu SA 70

Rysunek 38 Mapa cieplna przedstawiająca sumę wartości błędów względnych dla danej wartości parametru alfa dla algorytmu SA 70

Rysunek 39 Mapa cieplna przedstawiająca wartość procentową znalezienia najlepszych rozwiązań dla zadanej wartości parametru alfa dla algorytmu SA 71

10. Bibliografia

- [1] University of Waterloo 200 University Avenue West Waterloo, Ontario, N2L 3G1, [Online]. Available: <http://www.math.uwaterloo.ca/tsp/data/usa/tours.html>. [Accessed 18 11 2021].

- [2] Wydawnictwo Naukowe PWN ul. G. Daimlera 2, , „Watsona-Cricka-model-DNA,” Wydawnictwo Naukowe PWN, 1951. [Online]. Available: <https://encyklopedia.pwn.pl/haslo/Watsona-Cricka-model-DNA;3994315.html>. [Data uzyskania dostępu: 22 01 2022].
- [3] M. Maniecka, „Metody sekwencjonowania DNA,” Laboratoria.net, 26.03.2012.
- [4] Wydawnictwo Naukowe PWN , „Hybrydyzacja kwasów nukleinowych,” Wydawnictwo Naukowe PWN , [Online]. Available: <https://encyklopedia.pwn.pl/haslo/hybrydyzacja-kwasow-nukleinowych;3913455.html>. [Data uzyskania dostępu: 22 01 2022].
- [5] M. Radom, „Kombinatoryczne aspekty nieklasycznego sekwencjonowania DNA przez hybrydyzację,” dr hab. inż. Piotr Formanowicz, profesor PP, Poznań, 2011.
- [6] Z. L. Ratnesh Kumar, „Optimizing the Operation Sequence of a Chip Placement Machine Using TSP Model,” Department of Electrical Engineering University of Kentucky Lexington , KY 40506-0046, Kentucky, 2003.
- [7] Wydawnictwo naukowe PWN, „Heurystyka,” Wydawnictwo naukowe PWN, [Online]. Available: <https://encyklopedia.pwn.pl/haslo/heurystyka;4008452.html>. [Data uzyskania dostępu: 23 01 2022].
- [8] p. d. h. i. M. Kasprzak, „Zawansowane programowanie wykład : inne heurystyki,” Instytut Informatyki, Politechnika Poznańska, [Online]. Available: <http://www.cs.put.poznan.pl/mkasprzak/zp/ZP-wyklad3.pdf>. [Data uzyskania dostępu: 23 01 2022].
- [9] W. Z. A. Katedra Informatyki Stosowanej, „Inteligencja obliczeniowa: Heurystyki i metaheurystyki,” 11 01 2021. [Online]. Available: http://www.pi.zarz.agh.edu.pl/intObl/notes/IntObl_w2.pdf. [Data uzyskania dostępu: 11 01 2022].

- [10] The software quality company, [Online]. Available: <https://www.tiobe.com/tiobe-index/>. [Data uzyskania dostępu: 19 11 2021].
- [11] Python Software Foundation, „tracemalloc — Trace memory allocations,” Python Software Foundation, 01 05 2000. [Online]. Available: <https://docs.python.org/3/library/tracemalloc.html>. [Data uzyskania dostępu: 14 01 2022].
- [12] G. Rodola, „psutil documentation,” 27 01 2009. [Online]. Available: <https://psutil.readthedocs.io/en/latest/>. [Data uzyskania dostępu: 14 01 2022].
- [13] L. T. B. V. S. Z. O. O. W. POLSCE, „Ideapad Y700 (15),” LENOVO TECHNOLOGY B V SP Z O O , 11 01 2022. [Online]. Available: <https://www.lenovo.com/pl/pl/laptops/ideapad/y700-series/Ideapad-Y700-15/p/88IPY700618>. [Data uzyskania dostępu: 11 01 2022].
- [14] I. T. P. S. Z. O. O, „Processor Intel® Core™ i7-6700HQ (pamięć cache 6 MB, nawet do 3,50 GHz),” INTEL TECHNOLOGY POLAND SP Z O O, 11 01 2022. [Online]. Available: <https://www.intel.pl/content/www/pl/pl/products/sku/88967/intel-core-i76700hq-processor-6m-cache-up-to-3-50-ghz/specifications.html>. [Data uzyskania dostępu: 11 01 2022].
- [15] SAMSUNG ELECTRONICS POLSKA SP Z O O, „Samsung SSD 960 EVO 250GB,” SAMSUNG ELECTRONICS POLSKA SP Z O O, 11 01 2022. [Online]. Available: <https://www.samsung.com/pl/memory-storage/nvme-ssd/960-evo-nvme-m-2-ssd-250gb-mz-v6e250bw/>. [Data uzyskania dostępu: 11 01 2022].
- [16] CRITICAL SYSTEM SOLUTIONS SP Z O O, „Crucial MX500,” CRITICAL SYSTEM SOLUTIONS SP Z O O, 11 01 2022. [Online]. Available: <https://www.crucial.com/ssd/mx500/ct500mx500ssd1>. [Data uzyskania dostępu: 11 01 2022].

- [17] F. Goulart, „Repozytorium python-tsp,” Massachusetts Institute of Technology, 20 06 2020. [Online]. Available: <https://github.com/fillipe-gsm/python-tsp>. [Data uzyskania dostępu: 17 01 2022].
- [18] D. P. Williamson, „Anylysis of the Held-Karp Heuristic for the Traveling Salesman Problem,” Massachusetts, 1990.
- [19] administrator@ency.pl, „Programowanie dynamiczne,” Encyklopedia Algorytmów, 01 07 2017. [Online]. Available: http://algorytmy.ency.pl/artukul/programowanie_dynamiczne. [Data uzyskania dostępu: 24 01 2022].
- [20] C. L. R. R. C. S. T.H. Cormen, Wprowadzenie do algorytmów, Warszawa: Wydawnictwo Naukowe PWN, 2012.
- [21] administrator@ency.pl, „Algorytm Helda-Karpa,” Encyklopedia algorytmów, 07 07 2017. [Online]. Available: http://algorytmy.ency.pl/artukul/algorytm_helda_karpa. [Data uzyskania dostępu: 25 01 2022].
- [22] P. Szestało, „Badanie grafów Hamiltona z językiem Python,” Wydział Fizyki, Astronomi i Informatyki Stosowanej, Kraków, 2015.
- [23] Wikipedia, „Wyżarzanie,” [Online]. Available: <https://pl.wikipedia.org/wiki/Wy%C5%BCarzanie>. [Data uzyskania dostępu: 26 01 2022].
- [24] Wikipedia, „Symulowane Wyżarzanie,” [Online]. Available: https://pl.wikipedia.org/wiki/Symulowane_wy%C5%BCarzanie. [Data uzyskania dostępu: 26 01 2022].
- [25] „Symulowane Wyżarzanie,” Encyklopedia Algorytmów, [Online]. Available: http://algorytmy.ency.pl/artukul/symulowane_wyjarzanie. [Data uzyskania dostępu: 26 01 2022].

- [26] H. H. H. Thomas Stützle, „Analyzing the Run-time Behaviour of Iterated Local Search for the TSP,” University of British Columbia, University Libre de Bruxelles, Bruksela, Vancouver, 1999.
- [27] Wikipedia, „2-opt,” [Online]. Available: <https://en.wikipedia.org/wiki/2-opt>. [Data uzyskania dostępu: 26 01 2022].
- [28] Wikipedia, „Algorytm A*,” [Online]. Available: https://pl.wikipedia.org/wiki/Algorytm_A*. [Data uzyskania dostępu: 27 01 2022].
- [29] J. Raczyńska, „Algorytm A*,” [Online]. Available: <https://elektron.elka.pw.edu.pl/~jarabas/ALHE/notatki3.pdf>. [Data uzyskania dostępu: 27 01 2022].
- [30] Wikipedia, „Algorytm najbliższego sąsiada,” [Online]. Available: https://pl.wikipedia.org/wiki/Algorytm_najbli%C5%BCszego_s%C4%85siada. [Data uzyskania dostępu: 27 01 2022].
- [31] A. Y. A. Z. G. Gutin, „Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP,” Department of Computer Science, Royal Holloway, University of London, Londyn, 2001.
- [32] K. M., Sztuczne Życie. Algorytmy inspirowane biologicznie., Warszawa: Polska Akademia Nauk, 2008.
- [33] Wikipedia, „Algorytm genetyczny,” [Online]. Available: https://pl.wikipedia.org/wiki/Algorytm_genetyczny. [Data uzyskania dostępu: 28 01 2022].
- [34] M. Komosiński, „Czym jest sztuczne życie,” [Online]. Available: <http://www.alife.pl/czym-jest-sztuczne-zycie>. [Data uzyskania dostępu: 28 01 2022].
- [35] D. h. i. G. Dudek, „Sztuczna inteligencja wykład 15 Algorytmy genetyczne,” Wydział Elektryczny, Politechnika Częstochowska, 2014. [Online]. Available:

- https://gdudek.el.pcz.pl/files/SI/SI_wyklad15.pdf. [Data uzyskania dostępu: 29 01 2022].
- [36] J. Arabas, „Wykłady z algorytmów ewolucyjnych,” WNT, Warszawa, 2003.
- [37] d. i. W. Beluch, „Metody Heurystyczne Problem komiwojażera (TSP),” Instytut Mechaniki i Inżynierii Obliczeniowej, Wydział Mechaniczny Technologiczny, Politechnika Śląska, [Online]. Available: [http://www.imio.polsl.pl/Dopobrania/Cw%20MH%2007%20\(TSP\).pdf](http://www.imio.polsl.pl/Dopobrania/Cw%20MH%2007%20(TSP).pdf). [Data uzyskania dostępu: 28 01 2022].
- [38] G. a. R. L. J. David E, „Alleles, Loci, and Traveling Salesman Problem,” Department of Enineering Mechanics, The University of Alabama, Alabama, 1985.
- [39] D. L. Davis, „Job Shop Scheduling with Genetic Algorithm,” Bolt Beranek and Newman Inc., 1985.
- [40] L. D. Whitley, „Scheduling Problems and Traveling Salesman,” Colorado State University, Department of Computer Science, Colorado, 1989.
- [41] M. Yamamura, I. Ono i S. Kobayashi, „Emergent Search on Souble Circle TSPs using Subgour Exchange Crossover,” Tokyo Institute of Technology, Tokyo, 1996.
- [42] D. H. A. H. Darrel Whitley, „Tunneling between optima: partition crossover for the traveling salesman problem,” Colorado State Univeristy, Fort Collins, 2009.
- [43] Z. Michalewicz, „Algorytmy genetyczne + struktury danych = programy ewolucyjne,” WNT, Warszawa, 1996.
- [44] H. G, „Repozytorium biblioteki mlrose,” 01 2019. [Online]. Available: <https://github.com/gkhayes/mlrose>. [Data uzyskania dostępu: 28 01 2022].
- [45] guofei9987, „Repozytorium biblioteki scikit-opt,” 06 09 2019. [Online]. Available: <https://github.com/guofei9987/scikit-opt>. [Data uzyskania dostępu: 28 01 2022].

- [46] K. Bhatia, „Genetic Algorithms and the Traveling Salesman Problem,” University of California, San Diego, 1994.
- [47] V. M. A. C. Marco Dorigo, „An investigation of some properties of an "Ant algorithm",” Politecnico di Milano, Milano, Italy, 1992.
- [48] D. Błaszkiwicz, „Algorytmy Mrówkowe - wykład,” 11 05 2011. [Online]. Available: https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/alg_mrow.pdf. [Data uzyskania dostępu: 29 01 2022].
- [49] quofei9987, „Repozytorium biblioteki scikit-opt, algorytm ACO,” 11 09 2019. [Online]. Available: <https://github.com/quofei9987/scikit-opt/blob/master/sko/ACA.py>. [Data uzyskania dostępu: 29 01 2022].
- [50] E. R. Kennedy J., „Particle Swarm Optimalization,” Proc. IEEE Int. Conf. On Neural Networks, Piscataway, New Jersey, 1995.
- [51] A. Thevenot, „Partical Swarm Optimization (PSO) Visually Explained,” 21 12 2022. [Online]. Available: <https://towardsdatascience.com/particle-swarm-optimization-visually-explained-46289eeb2e14>. [Data uzyskania dostępu: 29 01 2022].
- [52] D. i. P. Foryś, „Zastosowanie metody roju cząstek w optymalnym projektowaniu elementów konstrukcji,” Wydawnictwo Politechniki Krakowskiej, Kraków, 2008.
- [53] K. Olszowy, „Aplikacja znajdująca najkrótszą drogę w supermarkecie,” Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, Kraków, 2017.